*Переклад українською - Арсеній Чеботарьов - Ніжин 2016*

# Akka Streams
# Вступ

## Мотивація

Шлях, яким ми споживаємо сервіси з інтернету сьогоді, включає багато примірників потокових даних, обоє, завантажених з сервісів, так само, як і надісланих до них, а також передачі даних типу точка-точка. Розглядати дані як потік елементів, замість як ціле, є дуже корисним, оскільки це співпадає зі шляхом, як комп'ютери надсилають та отримують їх (наприклад, через TCP), але також часто це необхідно, бо набори даних часто стають дуже довгими, щоб бути обробленими як ціле. Ми розпорошили обчислення та аналіз на великі кластери, та назвали їх "великими даними", де цілий принцип їх обробки є послідовне подання ціх даних як потока через деякі CPU.

Актори також можуть розглядатись як маючі справу з потоками: вони надсилають та отримують сервії повідомлень, щоб передавати знання (або дані) з одного місця в інше. Ми знайшли це виснажливим та схильним до помилок реалізувати всі відповідні виміри, щоб досягти стабільних потоків між акторами, оскільки на додаток до надсилання та отримки нам треба турбуватись, та не переповнити жодні буфери або поштові скриньки в процесі цього. Інша пастка в тому, що повідомлення акторів можуть бути втрачені, та мають бути повторно передані, в якому випадку потік має дірки на стороні отримувача. Коли маєте справу з потоками елементів фіксованого наданого типу, актори наразі не пропонують гарних статичних гарантій що того, що не відбулось помилок запису: в цьому випадку типо-безпечність може бути покращена.

З цієї причини ми вирішили запакувати рішення ціх проблем як  Akka Streams API. Призначення в наданні інтуітивного та безпечного шляху до формулювання налаштування обробки потоку, так що ми потім можемо їх ефективно обробляти, та з обмеженим використанням ресурсів  — більше немає OutOfMemoryErrors. Щоб досягти цього наші потоки потребують змоги обмежувати буферізацію, що вони задіють, та їм треба можливість уповільнювати прод'юсерів, якщо споживачі не можуть поспівати за ними. Ця можливість називається зворотнім тиском, та належить до ядра ініціативи Reactive Streams, до якої Akka є засновником. Для вас це означає, що складна проблема просування та реагування на зворотній тиск вже була вбудована в  розробку Akka Streams, так що ви маєте на одну річ менше, про яку не треба турбуватись; це також означає, що Akka Streams безпроблемно взаємодіє з усіма іншими реалізаціями Reactive Streams (де інтерфейси  Reactive Streams визначають взаємодію SPI, тоді як реалізації, як Akka Streams надають гарний користувацький API).

### Взаємодія з Reactive Streams

Akka Streams API повністю відділена від інтерфейсів Reactive Streams. Тоді як Akka Streams сфокусоване на формулюванні трансформацій з потоками даних, поле зору Reactive Streams є тільки визначити загальний механізм того, як пересувати дані між асинхронними межами безвтрат, буферизації або поглинання ресурсів.

Взаємовідносини між ціми двома в тому, що Akka Streams API спрямоване на кінцевого користувача, і в той же час реалізація Akka Streams внутрішньо використовує реалізацію інтерфейсів Reactive Streams для передачі даних між різними стадіями обробки. З цієї причини ви не знайдете жодної подібності між інтерфейсами Reactive Streams та Akka Streams API. Це співпадає з очікуваннями від проекту Reactive Streams, чиє первинне призначення є визначення інтерфейсів, таким чином, щоб різні реалізації потокв могли взаємодіяти; він не призначений щоб Reactive Streams описували API кінцевого користувача.

## Як читати ці документи

Обробка потоків є іншою парадігмою до моделі акторів, або до компонування Future, і таким чином треба уважно вивчити предмет, доки ви не почнете почуватись знайомими з інструментами цієї технології. Ця документація допоможе, та для отримання кращих результатів ми рекомендуємо наступний підхід:

- Прочитайте Інструкцію зі швидкого початку, щоб отримати відчуття від потоків, як вони виглядають, та що вони можуть робити.
- Ті, хто вивчає зверху до низу, можуть перейти до Принципи розробки, що стоять за Akka в цій точці.
- Ті, хто вивчає знизу до верху, можуть почуватись більш гарно, пошукавшу в Повареній книзі Streams.
- Для повного огляду вбудованих стадій обробки ви можете подивитись на таблицю в Огляді вбудованих стадій, та їх семантики
- Інші розділи можна читати послідовно, або як вимагають попередні кроки, в кожній поринаючи в специфічні теми.

# Інструкція зі швидкого початку

Потік звичайно починається з джерела, так що також там починає і Akka Stream. Перед тим, як ми створимо один з них, ми імпортуємо повний конплект потокових інструментів:

```
1. import akka.stream._
2. import akka.stream.scaladsl._
```

Якщо ви бажаєте виконати приклади кода під час читання цієї інструкції, вам також треба наступні імпортування:

```
1. import akka.{ NotUsed, Done }
2. import akka.actor.ActorSystem
3. import akka.util.ByteString
4. import scala.concurrent._
5. import scala.concurrent.duration._
6. import java.nio.file.Paths
```

Тепер ми будемо починати зі скоріше простого джерела, що видає цілі від 1 до 100:

```
1. val source: Source[Int, NotUsed] = Source(1 to 100)
```

Тип `Source` параметризовано двома типами: перший є типом елемента, що видає це джерело, та другий може сигналізувати, що виконання джерела продукує деяке додаткове значення (наприклад, мережеве джерело може провадити інформацію щодо прив'язаного порта або адреси). Якщо додаткова інформація не продукується, використовується `akka.NotUsed` — та проситий диапазон цілих підпадає під цю категорію.

Маючи створеним це джерело, означає, що ми маємо опис того, як видати 100 натуральний чисел, та це джерело ще не активне. Щоб отримати ці числа, ми маємо запустити його:

```
1. source.runForeach(i => println(i))(materializer)
```

Цей рядок завершить джерело за допомогою функції споживача — в прикладі ми просто друкуємо числа на консолі — та передаємо цей малий потік до актора, що виконує його. Ця активація позначається як слово “run”, що є частиною назви метода; існують інші методи, що виконують потокі Akka Streams, та всі вони слідують цьому шаблону.

Ви можете здивуватись, де створюється актор, що виконує потік, і ви, можливо питаєте себе, що означає цей `materializer`. Щоб отримати це значення, спочатку ми створюємо систему акторів:

```
1. implicit val system = ActorSystem("QuickStart")
2. implicit val materializer = ActorMaterializer()
```

Існують інші шляхи створити materializer, наприклад, з `ActorContext`, коли потоки використовуються в самих акторах. `Materializer` є фабрикою для пристроїв виконання потоку, це те, що змушує потік рухатись — вам прямо зараз не треба турбуватись щодо жодних деталей, крім того, що вам знадобиться один такий матеріалізатор, для виклику методів `run` на `Source`. Матеріалізатор підхоплюється неявно, якщо він не вказаний в аргументах метода `run`, що ми будемо робити в в подальшому.

Гарна річ щодо Akka Streams полягає в тому, що `Source` є тільки описом того, що потрібно використовувати, та як проект архітектора, це можна використовувати повторно, вбудоване в інший дизайн. Ми можемо обрати перетворення джерела цілих та записати їх до файлу:

```
1. val factorials = source.scan(BigInt(1))((acc, next) => acc * next)
2.
3. val result: Future[IOResult] =
4.   factorials
```

```
    .map(num => ByteString(s"$num\n"))
    .runWith(FileIO.toPath(Paths.get("factorials.txt")))
```

Спочатку ми використовуємо комбінтор `scan` для виконання обчислень над цілим потоком: починаємо з числа 1 (`BigInt(1)`) ми множимо кожне з вхідних чисел, одне за одним; операція scan видає початкове значення, та потім кожний обчислений результат. Це дає серію чисел факториала, що ми зберігаємо як `Source` для майбутнього використання — важливо пам'ятати, що наразі нічого не обчислюється, це тільки опис того, що ми бажаємо обчислити, коли ми виконаємо потік. Потім ми конвертуємо отриману серію чисел на потік об'єктів `ByteString`, що описують рядки текстового файла. Цей потік потім виконується, через додавання файла як отримувача даних. В термінології Akka Streams це називається приймач, `Sink`. `IOResult` є типом, що повертають операції IO в Akka Streams, щоб сказати, як багато байтів або елементів було спожито, та чи потік завершився нормально або з виключенням.

## Поторно задіяні частини

Одна з гарних речей в Akka Streams — ата дещо, чого не пропонують інші поточні бібліотеки — це те, що не тільки джерела можуть бути задіяні як плани, але також і всі інші елементи. Ми можемо взяти `Sink`, що пише в файл, поставити попереду кроки обробки, потрібні для отримання елементів `ByteString` з надходячих рядків, і так само запакувати це як повторно задіяну частину. Оскільки мова для написання цїх потоків завжди слідує зліва направо (як звичайна англійська), нам треба початкова точка, що як джерело, але з "відкритим" входом. В Akka Streams це називається `Flow`:

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

Починаючи з потока рядків, ми конвертуємо кожний на `ByteString`, та потім подаємо до вже знайомого приймача `Sink`, що пише в файл. Отримана схема є `Sink[String, Future[IOResult]]`, що означає, що він сприймає рядкі як свій вхід, та під час матеріалізації він створює додаткову інформацію типу `Future[IOResult]` (коли зціплюються операції на `Source` або `Flow`, тип допоміжної інформації — що називається “матеріалізоване значення”— надаєть в самій лівій початковій точці; осклыіки ми бажаємо залишити те, що пропонує нам приймач `FileIO.toPath`, нам треба сказати `Keep.right`).

Ми можемо використати новий та блискучий `Sink`, що ми тільки но створили, приєднавши його до нашого джерела `factorials` — після невеликої адаптації для перетворення чисел на рядки:

```
factorials.map(_.toString).runWith(lineSink("factorial2.txt"))
```

## Обробка на основі часу

Перед тим, як ми поглянемо на більш складний приклад, ми дослідимо потокову природу того, що може робити Akka Streams. Починаючи з джерела `factorials` ми трансформуємо потік, зібравши його разом з іншим потоком, що представлений `Source`, що видає числа з 0 до 100: перше число, що видається джерелом `factorials` є факториал нуля, другий є факториалом одиниці, і так далі. Ми комбінуємо ці два через формування рядків типу `"3! = 6"`.

```
1. val done: Future[Done] =
2.   factorials
3.     .zipWith(Source(0 to 100))((num, idx) => s"$idx! = $num")
4.     .throttle(1, 1.second, 1, ThrottleMode.shaping)
5.     .runForeach(println)
```

До цього часу всі операції були незалежні від часу, та моги бути виконані в тому ж самому стилі на обмежених колекціях елементів. Наступний рядок демонструє, як, насправді, ми маємо справу з потоками, що можуть надходити з певною швидкістю: ми використовуємо комбінатор `throttle` для уповільнення потоку до 1 елементу на секунду ( `1` секунда в списку аргументів є максимальним прискоренням, що ми бажаємо дозволити — передача `1` означає, що перший елемент проходить безпосередньо, та потім другий має чекати одну секунду, і так далі).

Якщо ви виконаєте цю програму, ви побачите друк одного рядка в секунду. Заслуговує згадки один аспект, що не є беспосередньо наочним: якщо ви спробуєте та встановите потоки для продукування мільярд чисел кожний, тоді ви помітите, що ваша JVM не впаде з OutOfMemoryError, та навіть ви помітите, що виконання потоків відбувається в фоні, асинхронно (це та причина, чому допоміжна інформація провадиться як `Future`). Секрет, що змушує це робити, є те, що Akka Streams неявно реалізує розширений контроль потоку, всі комбінатори поважають зворотній тиск. Це дозволяє комбінатору throttle сигналізувати всім своїм джерелам даних, що він приймає елементи тільки з певною швидкістю — коли входяций потік вищий, ніж один за секунду, комбінатор throttle буде накладати *зворотній тиск*.

Це в основному все щодо Akka Streams — прикрашене тим, що існують десятки джерел та приймачів, та багато більше комбінаторів перетворень потоку, з яких можна обирати. Також дивіться Огляд вбудованих стадій та їх семантика.

# Reactive Tweets

Типовим прикладом обробки потока є споживання живого потоку даних, коли ми бажаємо виділити або агрегувати деякі дані з нього. В цьому прикладі ми розглянемо споживання потоку твітів, та виділення з нього інформації щодо Akka.

Також ми розглянемо проблему, що постає для всіх неблокуючих поточинх рішень: *"Що, якщо підписчик є досить повільним для споживання живого потоку даних?"*. Традиційне рішення є буферизація елементів, але це може — та звичайно буде — викликати переповнення буфера та нестабільність систем.

Замість цього Akka Streams покладається на внутрішні сигнали зворотнього тиску, що дозволяють контролювати, що повинно відбуватись в такому сценарії.

Ось модель даних, з якою ми будемо працювати під час роботи над цім прикладом:

```
1. final case class Author(handle: String)
2.
3. final case class Hashtag(name: String)
4.
5. final case class Tweet(author: Author, timestamp: Long, body: String) {
6.   def hashtags: Set[Hashtag] =
7.     body.split(" ").collect { case t if t.startsWith("#") => Hashtag(t) }.toSet
8. }
9.
10. val akka = Hashtag("#akka")
```

**Зауваження**

Якщо ви бажаєте отримати огляд використовуваного словника, перед тим. як поринути в саме приклад, подивіться на розділи Ключові концепції та Визначення та виконання потоків в документації, та потім повертайтесь до цього прикладу, щоб побачити всі частини разом в простому застосуванні-прикладі.

# Трансформація та споживання простих потоків

Застосування-приклад, який ми розглядаємо, є простий потік Twitter, з якого ми бажаємо виділити певну інформацію, як для приклада шукаючи всі гачки твіттера до користувачів, що твітують щодо `#akka`.

Щоб підготувати наше оточення через створення `ActorSystem` та `ActorMaterializer`, що будуть відповідальні за матеріалізацію та виконання потоків, ми зараз створимо два неявні об'єкти:

```
1. implicit val system = ActorSystem("reactive-tweets")
2. implicit val materializer = ActorMaterializer()
```

`ActorMaterializer` опціонально може сприймати `ActorMaterializerSettings`, що може бути використаний для визначення властивостей матеріалізації, як розмір буфера по замовчанню (також дивіться Буфери для асинхронних стадій), використовуваний конвеєром диспечер, тощо. Вони можуть бути перевизначені за допомогою `withAttributes` на `Flow`, `Source`, `Sink` та `Graph`.

Давайте уявімо, що ми вже маємо готовий потік з твітів. В Akka це виражається як `Source[Out, M]`:

```
1. val tweets: Source[Tweet, NotUsed]
```

Потоки завжди починають текти від `Source[Out,M1]`, потім вони продовжуються в елементах `Flow[In,Out,M2]`, або більш складних елементах графів, щоб кінець кінцем бути спожитими `Sink[In,M3]` (поки ми ігноруємо параметри типів `M1`, `M2` та `M3`, вони не релевантні до типів елементів, що продукуються/споживаються ціма класами – вони є "матеріалізованими типами", про що ми поговоримо нижче).

Операції повинні виглядати знайомими для кожного, хто використовував бібліотеку Scala Collections, однак вони оперують на потоках, та не на колекціях даних (що є дуже важлива різниця, бо деякі операції мають сенс тільки в потоках, та навпаки):

```
1. val authors: Source[Author, NotUsed] =
2.   tweets
3.     .filter(_.hashtags.contains(akka))
4.     .map(_.author)
```

Нарешті, щоб матеріалізувати та виконати обчислення потока, нам треба приєднати Flow до `Sink`, що зробить Flow працюючим. Найпростіший шлях зробити це є виклик `runWith(sink)` на `Source`. Для зручності вже визначені декілька загальних методів на об'єкті-компанйоні `Sink`. Тепер давайте просто роздрукуємо кожного автора:

```
1. authors.runWith(Sink.foreach(println))
```

або використовуючи скорочену версію (що визначена тільки для найбільш популярних Sink, таких як `Sink.fold` та `Sink.foreach`):

```
1. authors.runForeach(println)
```

Матеріалізація та виконання потоку завжди потребує, щоб в неявному полі зору був `Materializer` (або переданий явно, таким чином: `.run(materializer)`).

Повний фрагмент кода виглядає так:

```
1. implicit val system = ActorSystem("reactive-tweets")
2. implicit val materializer = ActorMaterializer()
3.
4. val authors: Source[Author, NotUsed] =
5.   tweets
6.     .filter(_.hashtags.contains(akka))
7.     .map(_.author)
8.
9. authors.runWith(Sink.foreach(println))
```

## Сплющення послідовностей в потоках

В попередньому розділі ми робили з відношеннями 1:1 між елементами,  що є найбільш загальним випадком, але інколи ми можемо захотіти відобразити з одного елемента до декількох елементів, та отримати "сплющений" потік, так само, як робить `flatMap` з Scala Collections. Щоб отримати сплющений потік з хештегів з нашого потоку твітів, ми можемо використати комбінатор `mapConcat`:

```
1. val hashtags: Source[Hashtag, NotUsed] = tweets.mapConcat(_.hashtags.toList)
```

**Зауваження**

Ім'я `flatMap` було навмисне уникнуте через його близькість до for-осяжностей та композиції монад. Це проблематично з двох причин: перше, сплющення через конкатенацію є часто небажаним при обробці обмеженого потоку, через ризик глухого кута (де злиття є переважною стратегією), та, по-друге, закон монад не буде виконаний в нашій реалізації flatMap (через проблеми з життєспроможністю).

Будь ласка зауважте, що `mapConcat` потребує наданої функції для повертання обмеженої колекції (`f:Out=>immutable.Seq[T]`), в той час, коли `flatMap` має оперувати з потоками, що надходять весь час.

## Широкомовлення потоків

Тепер, скажімо, ви бажаєте зберігти всі хештеги, а також всі імена авторів з цього одного живого потоку. Наприклад, ми хочемо записати всі зачіпки авторів до файлу на диск. Це означає, що ми маємо розділити потік-джерело на два потоки, що будуть обробляти запис цих окремих файлів.

Елементи, що можуть використовуватись для формування таких структурних розгалужень в Akka Streams називають "поєднаннями". Одне з таких, що ми використаємо в цьому прикладі, називається `Broadcast`, та просто передає елементи на вхідному порті на всі свої вихідні порти.

Akka Streams навмисне роділяє лінійні структури потоку (Flow), та нелінійні, як розгалуження (Graphs), щоб запропонувати найбільш зручний API для обох цих випадків. Графи можуть виражати довільно складні потоки, ціною того, що вони не читаються так само дружньо, як перетворення колекцій.

Графи конструюються за допомогою `GraphDSL` таким чином:

```
1. val writeAuthors: Sink[Author, Unit] = ???
2. val writeHashtags: Sink[Hashtag, Unit] = ???
3. val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
4.   import GraphDSL.Implicits._
5.
6.   val bcast = b.add(Broadcast[Tweet](2))
7.   tweets ~> bcast.in
8.   bcast.out(0) ~> Flow[Tweet].map(_.author) ~> writeAuthors
9.   bcast.out(1) ~> Flow[Tweet].mapConcat(_.hashtags.toList) ~> writeHashtags
10.  ClosedShape
11. })
12. g.run()
```

Як ви можете бачити, в `GraphDSL` ми використовуємо неявний побудовник графів `b` щоб змінно конструювати граф з використанням "оператора гачка" `~>` (також читається є "з'єднати" або "через", або "до"). Оператор провадиться неявно через імпорт `GraphDSL.Implicits._`.

`GraphDSL.create` повертає `Graph`, в цьому випадку `Graph[ClosedShape, Unit]`, де `ClosedShape` означає, що це *повністю з'єднаний граф* або "замкнений" - немає непід'єднаних входів та виходів. Оскільки він замкнений, можливо трансформувати граф в `RunnableGraph` з використанням `RunnableGraph.fromGraph`. Виконуваний граф може бути виконаний через `run()` для матеріалізації з нього потоку.

Обоє, `Graph` та `RunnableGraph` є *незмінними, потік-безпечними, та вільно розподільними*.

Граф також може мати одну з декількох інших форм, з одним або більше непід'єднаними портами. Маючи непід'єднані порти виражає граф, що є *частковим графом*. Концепція коло композиції та вкладання графів в великі структури виражена в деталях Модулярність, композиція та ієрархія. Також можливо обернути граф зі складними обчисленнями як Flow, Sink або Source, що буде виражене в деталях Конструювання Source, Sink та Flow з часткових графів.

## Зворотній тиск в дії

Одна з головних переваг Akka Streams в тому, що вони *завжди* пропагують інформацію зворотнього тиску від збирачів потоку Sink (підписчики) до їх джерел Source (публіканти). Це не опціональна можливість, та вона ввімкнена весь час. Щоб вивчити більше щодо протокола зворотнього тиску, що використовується в Akka Streams, та у всіх інших реалізаціях, сумісних з Reactive Streams, читайте Пояснення зворотнього тиску.

Типова проблема, з якою стикаються застосування, на кшталт цього (не використовуючи Akka Streams), полягає в тому, що вони не в змозі обробити вхідні дані досить швидко, або тимчасово, або за дизайном, та будуть починати буферизувати вхідні дані, доки не залишиться вільного простіру для буфера, що призведе до або `OutOfMemoryError`, або до іншої суттєвої деградації відповідальності сервера. За допомогою Akka Streams буферизація може і мусить бути оброблена явно. Наприклад, якщо ми зацікавлені тільки в "*найбільш останніх твітах, з буфером в 10 елементів*", це може бути виражене з використанням елементу `buffer`:

```
1. tweets
2.   .buffer(10, OverflowStrategy.dropHead)
3.   .map(slowComputation)
4.   .runWith(Sink.ignore)
```

Елемент `buffer` приймає явне та необхідне `OverflowStrategy`, що визначає, як буфер повинен реагувати, коли надходить інший елемент, коли він вже заповнений. Запроваджені стратегії включають відкидання цілого буфера, сигналізацію про помилку, тощо. Переконайтесь, що обрали стратегію, що найкраще пасує до вашого випадка.

## Матеріалізовані значення

До тепер ми обробляли тільки дані з використанням Flows, та споживаючи їх в деякому різновиді зовнішнього Sink - було це друком значень, або зберіганням в деякій зовнішній системі. Однак іноді ми можемо бути зацікавлені в деякому значенні, що може бути отримане від матеріалізованого конвеєра обробки. Наприклад, ми бажаємо знати, як багато твітів ми обробили. Хоча це питання не таке очевидне, коли отримати відповідь для безкінечного потоку твітів (один зі шляхів відповісти не це запитання  налаштуванні потоків може бути створення потоку з лічильників, описаних як "*аж до тепер,* ми обробили N твітів"), але загалом можливо мати справу з кінечними потоками, та отримати гарний результат, такий, як загальний відлік елементів.

Спершу, давайте напишемо такий підрахунок елементів, з використанням `Sink.fold`, та побачимо, як будуть виглядати типи:

```
1. val count: Flow[Tweet, Int, NotUsed] = Flow[Tweet].map(_ => 1)
2.
```

```
3. val sumSink: Sink[Int, Future[Int]] = Sink.fold[Int, Int](0)(_ + _)
4.
5. val counterGraph: RunnableGraph[Future[Int]] =
6.   tweets
7.     .via(count)
8.     .toMat(sumSink)(Keep.right)
9.
10. val sum: Future[Int] = counterGraph.run()
11.
12. sum.foreach(c => println(s"Total tweets processed: $c"))
```

Спершу, ми приготували повторно використовуваний `Flow`, що буде змінювати кожний надходящий твіт на значення `1`. Ми використовуємо це, щоб скомбінувати це з `Sink.fold`, що буде підсумовувати всі `Int` елементи потоку, та робити результат доступним як `Future[Int]`. Далі ми з'єднаємо потік `tweets` до `count` за допомогою `via`. Нарешті, ми з'єднаємо Flow до попередньо підготованого Sink з використанням `toMat`.

Пам'ятаєте ці містичні типи параметрів на `Source[+Out, +Mat]`, `Flow[-In, +Out, +Mat]` та `Sink[-In, +Mat]`? Вони представляють тип значень, чиї частини обробки повертаються, коли матеріалізовані. Коли ви з'єднаєте їх разом, ви можете явно скомбінувати їх матеріалізовані значення. В нашому прикладі ми використали попередньо визначену функцію `Keep.right`, що каже реалізації турбуватись тільки про матеріалізований тип сцени, що наразі приєднаний зправа. Матеріалізований тип `sumSink` є `Future[Int]`, та оскільки використовується `Keep.right`, результуючий `RunnableGraph` також має параметр тиу `Future[Int]`.

Цей крок *доки не матеріалізує* конвеєр обробки, він тільки підоготував опис нашого Flow, що тепер під'єднаний до Sink, та до якого можна застосувати `run()`, як вказано в його типі : `RunnableGraph[Future[Int]]`. Далі ми викликаємо `run()`, що використовує неявний `ActorMaterializer` для матеріалізації та виконання Flow. Значення, що повертається викликом `run()` на `RunnableGraph[T]` є тип `T`. В нашому випадку цей тип `Future[Int]`, який, коли завершуєтсья, буде містити загальну довжину нашого потоку `tweets`. В випадку збою потоку, ця футура бавершиться з Failure.

`RunnableGraph` може повторно використовуватись та матеріалізуватись декілька разів, оскільки це тільки "схема" потока. Це означає, що якщо ми матеріалізуємо потік, наприклад, такий, що матеріалізує потік твітів на пртязі хвилини, матеріалізовані значення для ціх двох матеріалізацій будуть різними, як проілюстровано в цьому прикладі:

```
1. val sumSink = Sink.fold[Int, Int](0)(_ + _)
2. val counterRunnableGraph: RunnableGraph[Future[Int]] =
3.   tweetsInMinuteFromNow
4.     .filter(_.hashtags contains akka)
```

```
    .map(t => 1)
    .toMat(sumSink)(Keep.right)

// матеріалізуємо потік один раз вранці
val morningTweetsCount: Future[Int] = counterRunnableGraph.run()
// та ще раз ввечері, повторно використовуючи потік
val eveningTweetsCount: Future[Int] = counterRunnableGraph.run()
```

Багато елементів в  Akka Streams провадять матеріалізовані значення, що можуть використовуватись для отримання або результатів обчислень, або керувати, що будуть обговорені в деталях в Матеріалізації потока. Підсумовуючи цей розділ, тепер ми знаємо, що відбувається за лаштунками, коли ми виконуємо ці однорядкові програми, що еквівалентні до багаторядкових версій вище:

```
val sum: Future[Int] = tweets.map(t => 1).runWith(sumSink)
```

**Зауваження**

`runWith()` є зручним методом, що автоматично ігнорує матеріалізоване значення любих інших стадій, за винятком доданої самою `runWith()`. В прикладі вище він транслює до `Keep.right`, як комбінатора для матеріалізованих значень.

# Принципи побудови Akka Streams

Це зайняло довгий час, докі ми не були досить задоволені виглядом та відчуттям від API та архитектури реалізації, та доки не були направлені інтуіцією фази розробки, було багато пошукових досліджень. Цей розділ деталізує добутки та кодує їх в набір принципів, що виникали в цьому процесі.

**Зауваження**

Як детально описано у вступі, майте не увазі, що Akka Streams API повністю відділене від інтерфейсів Reactive Streams, що є тільки деталлю реалізації, як передавати дані потока між окремими стадіями обробки.

## Що можуть очікувати користувачі Akka Streams?

Akka побудовано на свідоммоу рішені запропонувати API, що є мінімальними та узгодженими — на відміну від простих та інтуітивних. Кредо полягає в тому, що ми обираємо явність перед магією, та якщо ми пропонуємо можливість, вона має робити завжди, без винятків. Інший спосіб сказати те саме, це те, що ми мінімізували число правил, що має вивчити користувач, замість намагання утримати правила близькими до того, що, за нашими міркуваннями, очікує користувач.

Маючи це, наступні принципи були реалізовані в Akka Streams:

- всі можливості явні в API, ніякої магії
- виняткова конпонованість: зкомбіновані шматки зостаються функцією кожної частини
- виключна модель домена розподіленох обмеженої обробки потоків

Це означає, що ми провадимо всі інструменти, потрібні для виразу любих топологій обробки потоків, що ми моделюємо всі основні аспекти цього домену (зворотній тиск, буферизація, трансформації, відновлення після збою, тощо), та що те, що будує користувач, буде повторно використане в більшому контексті.

## Akka Streams не надсилають пропащі елементи потоку до офісу загублених листів

Одне віжливе слідоцтво з запропонування тільки  можливостей, на які можна покладатись, є те обмеження, що бібліотека Akka Streams не може запевнити, що всі  об'єкти, передані на топологію обробки, будуть оброблені. Елементи можуть загубитись через декілька причин:

- звичайний користувацький код може спожити один елемент на етапі map(...), та спродукувати повністі інший як результат
- загальні операції з потоками навмисне відкидають елементи, тобто це take/drop/filter/conflate/buffer/…
- збій потоку знесе потік без очікування завершення обробки, та всі підвішені елементи будуть відкинуті
- завершення потоку буде просувати догори (тобто, від оператора take), що призаеде до завершення обробки горішніх кроків, без обробки всіх своїх вхідних елементів

Це означає, що надсилання JVM о'бєктів в потік, що має бути очищений, буде потребувати від користувача перевірки, що це відбувається за межами механізмів Akka Streams (тобто, очищаючи їх після таймаута, або коли їх результати переглянуті на виході потоку, або використовуючи інші важелі, як фіналізація, тощо).

## Отримані обмеження реалізації

Компонованість тягне за собою повторне використання часткових топологій потоку, що веде нас до піднесеного підходу опису потоків даних як (часткових) графів, що можуть діяти як композитні джерела, потоки (відомі як канали) та приймачі даних. Ці будівельні блоки потім будуть вільно розподілені, з можливістю скомбінувати їх, щоб сформувати більші графи. Презентація ціх шматків, таким чином, мусить бути незмінним планом, що матеріалізується в явний спосіб, щоб почати обробку потоку. Результуючий рушій обробки потоків потім є незмінним, в сенсі що він має сталу топологію, що описана в схемі. Динамічні мережі мають бути змодельовані з явним використоанням інтерфейсів Reactive Streams для підключення різних рушиїв разом.

Процес матеріалізації часто буде створювати специфічні об'єкти, що корисні для взаємодії з рушієм обробки під час роботи, наприклад, для її завершення, або для виділення метрик. Це означає, що функція матеріалізації продукує результат під назвою *матеріалізоване значення графа*.

## Взаємодія з іншими реалізаціями Reactive Streams

Потоки Akka Streams повністю реалізують специфікацію Reactive Streams, та взаємодіють з усіма іншими відповідними реалізаціями. Ми обрали повністю відділити інтерфейси Reactive Streams від користувацького API, оскільки ми розглядаємо їх як SPI, що не нецілені на кінцевого споживача. Щоб отримати `Publisher` або `Subscriber` з топології Akka Stream, має використовуватись відповідний елемент `Sink.asPublisher` або `Source.asSubscriber`.

Всі процесори потока Processor, що продукуються по замовченню матеріалізацією Akka Streams, обмежені мати єдиний Subscriber, додаткові підписчики будуть відхилені. Причиною для цього є те, що топології потоків, що описуються з використанням нашого DSL, ніколи не потребують від сторони паблішера поведінки розгалуження, всі розгалуження робляться з використанням явних елементів, як `Broadcast[T]`.

Це означає, що має використовуватись `Sink.asPublisher(true)` (для підтримки розгалуження), де потрібна широкомовна поведінка для взаємодії з іншими реалізаціями Reactive Streams.

## Що мають очікувати користувачі поточної бібліотеки?

Ми очікуємо, що на базі Akka Streams будуть побудовані бібліотеки, та фактично Akka HTTP є одна з таких прикладів, що живе в самому проекті Akka. Щоб дозволити користувачам мати вигоду від принципів, що описані вище в Akka Streams, встановлені наступні правила:

> - бібліотеки повинні провадити своїх користувачів повторно використовуваними частинами, тобто, показувати фабрики, що повертають графи, дозволяючи повну компонованість
> - бібліотеки можуть опціонально та додатково провадити можливості, що споживають та матеріалізують графи

Міркування поза першим правилом в тому, що компонованість буде зруйнована, якщо різні бібліотеки тільки приймають графи, та очікується, що вони матеріалізують їх: використання двох з них разом буде неможливим, бо матеріалізація може відбутись тільки один раз. Як наслідок, функціональність бібліотеки має бути виражена так, що матеріалізація могла бути виконана користувачем, за межами контролю бібліотеки.

Друге правило дозволяє бібліотеці додатково провадити гарний цукор для загальних випадків, прикладом чого є Akka HTTP API, що провадить метод `handleWith` для зручної матеріалізації.

**Зауваження**

Один важливий наслідок з цьогов полягає в тому, що опис повтороно описаного потоку не може бути прив'язаний до “живих” ресурсів, любі вз'язки до розміщення таких ресурсів мають бути відкладені до часу матеріалізації. Прикладами "живих" ресурсів є вже існуючі TCP з'єднання, широкомовний

Publisher, тощо; TickSource не підпадає в цю категорію, якщо таймер створений тільки під чс матеріалізації (як це є в випадку нашої реалізації).

Виключення з цього правила повинні бути гарно виважені та ретельно задокументовані.

### Отримані обмеження реалізації

Akka Streams має дозволяти виражати любу утіліту обробки в термінах незмінного плана. Найбільш загальними будівельними блоками є:

- Source: дещо, з тільки одним вихідним потоком
- Sink: дещо, з тільки одним вхідним потоком
- Flow: дещо, з тільки рівно одним входом та виходом
- BidiFlow: дещо, з точно двома вхідними, та двома вихідними потоками, що концептуально поводиться як два Flows в різних напрямках
- Graph: запакована топологія обробки потоку, що показує певний набір вхідних та вихідних портів, що характеризовані як об'єкт типа `Shape`.

**Зауваження**

Джерело, що видає потік потоків, є все ще звичайне джерело Source, тип елементів, які воно продукує, не відіграє ролі в статичній топології потоку, що він виражає.

## Різниця між помилками Error та відмовами Failure

Початкова точка цієї дискусії є визначення, надане Reactive Manifesto. Переведене на потоки, це означає, що помилка (error) допустима в потоці як нормальний елемент даних, тоді як відмова означає, що сам потік схибив та склався. В конкретних термінах, на рівні елементів даних інтерфейса Reactive Streams (включаючи помилки) сигналюється через `onNext`, тоді як відмови підіймають сигнал `onError`.

**Зауваження**

На жаль, ім'я метода для повідомлення про *відмову* до Subscriber назване `onError` з історичних причин. Завжди пам'ятайте, що інтерфейси Reactive Streams (Publisher/Subscription/Subscriber) моделюють низькорівневу інфраструктуру для передачі потоків між одиницями виконання, та помилки на цьоу рівні є в точності відмови, про які ми кажемо на вищому рівні, що можелюється в Akka Streams.

Є тільки обмежена підтримка для трактування `onError` в Akka Streams, порівняно з операторами, що доступні для трансформації елементів даних, що є навмисним в дусі попереднього параметра. Оскільки `onError` сигналізує, що потік згортається, його семантика доставки не така сама, як для завершення

потоку: стадії трансформації любого сорту буде тільки згортатись разом з потоком, можливо все ще зберігаючи елементи в неявних і явних буферах. Це означає, що елементи даних, випущені до збою, все що можуть бути втрачені, якщо `onError` наздожене їх.

Здатність збоїв просуватись скоріше, ніж елементи даних, є базовою для зривання потоків, що є під зворотнім тиском — особливо оскільки зворотній тиск може бути в режимі збоя (тобто, відключаючи буфери нагору, що потім перериваються, бо не можуть робити більше нічого іншого, або в разі виникнення глухого блокування).

### Семантика відновлення потока

Елемент відновлення (тобто люба трансформація, що абсорбує сигнал `onError`, та перетворює це на можливо більше елементів даних, за якими слідує нормальне завершення потока) діє як перегородка, що обмежує згортання потоку до даного регіону топології потоку. В згорнутому регіоні буферизовані елементи можуть бути втрачені, але зовні збій не має впливу.

Це робить в тому ж стилі як і вираз `try`–`catch`: маркується регіон, в якому перехоплюються виключення, але точний розмір кода, що був пропущений в цьому регіоні в разі збою, не може бути визначений точно  — має значення розташування тверджень.

# Основи роботи з Flows

## Ключові концепції

Akka Streams є бібліотекою для обробки та передачі послідовності елементів з використанням обмеженого простору буферів. Ця остання властивість є те, на що ми посилаємось як на *необмеженість*, та є визначною властивістю Akka Streams. Перекладаючи на повсяденну мову це можне виразити як ланцюжок (або, як ми побачимо пізніше, графи) оброблюючих сутностей, кожна виконується окремо (та, можливо, конкурентно) від інших, при цьому беферизуючи тільки обмежене число число елементів в кожний окремий момент. Ця властивість обмежених буферів є одною з різниць від моделі акторів, де кожний актор зіичайно необмежену поштоку скриньку, або обмежену з відкиданням. Сутності обробки Akka Stream мають обмежені "поштові скриньки", що не гублять елементи.

Перед тим, як ми рушимо далі, давайте визначимо деякі базові терміни, що будуть використовуватись на протязі всієї документації:

**Потік (Stream)**
: Активний процес, що включає пересування та трансформацію даних.

**Елемент (Element)**
: Елемент є одницею обробки потоків. Всі операції трансформаційй та передачі елементів від вхідного до вихідного потоків. Розміри буферів завжди виражені як число елементів, незалежно від дійсного розміру елементів.

**Зворотній тиск (Back-pressure)**
: Засіб для керування потоком, шлях для споживачів даних повідомляти прод'юсера щодо власної доступності, ефективно уповільнюючи прод'юсера апстриму, щоб задовільнити власну швидкість споживання. В контексті Akka Streams зворотній тиск завжди розуміється як *неблокуючий* та *асинхронний*.

**Неблокуючий**
Спосіб, в який певна операція не призупиняє прогрес викликаючого потоку, навіть якщо це займає довгий час, щоб завершити запитану операцію.
**Граф**
Опис топології обробки потока, визначена як шляхи, якими мають слідувати елементи під час обробки потоку.
**Стадія обробки**
Загальне ім'я для всіх будівельних блоків, що складають Граф. Приклади стадії обробки будуть операції, такі як `map()`, `filter()` власні `GraphStage` та поєднання графів, такі, як `Merge` або `Broadcast`. Для повного списку вбудованих стадій обробки дивіться Огляд вбудованих стадій та їх семантика

Коли ми кажемо про *асинхронний, неблокуючий зворотній тиск,* ми маємо на увазі, що стадії обробки, доступні в Akka Streams, не будуть використовувати блокуючі виклики, але асинхронну передачу повідомлень, для обміну повідомленнями один з одним, та вони будуть використовувати асинхронні механізми для уповільнення швидкого прод'юсера без блокування потока. Цей дизайн дружній до пулів потоків, оскільки сутності, що мають чекати (швидкий прод'юсер, що очікує повільного споживача) не будуть блокувати потік, але можуть повернути його для подальшого використання в пул потоків.

# Визначення та виконання потоків

Лінійна обробка каналів може бути виражена в Akka Streams з використанням наступних головних абстракцій:

**Джерело (Source)**
Стадія обробки, що має *рівно один вхід*, що випускає елементи даних, коли наступні стадій обробки даунстриму готові отримати їх.
**Приймач (Sink)**
Стадія обробки з *рівно одним входом*, що запитує та сприймає елементи даних, можливо уповільнюючи прод'юсера елементів апстріму
**Потік (Flow)**
Стадія обробки, що має *точно один вхід та вихід*, що поєднує свій ап- та даунстрім, трансформуючи елементи даних, що проходять через нього.
**Виконуваний граф (RunnableGraph)**
Flow, що має під'єднаними обоє кінці до Source та Sink відповідно, та готовий до `run()`.

Можливо приєднати `Flow` до `Source`, що утворить композитне джерело, та також можливо поставити `Flow` перед `Sink`, щоб отримати новий приймач. Після того, як потік відповідно завершений, маючи обоє, джерело та приймач, він буде представлений типом `RunnableGraph`, що вказує, що він готовий до виконання.

Важливо пам'ятати, що навіть після конструювання `RunnableGraph` через поєднання всіх: джерела, приймача, та всіх стадій обробки, дані не будть текти по ньому, доки він не буде матеріалізований. Матеріалізація є процесом розміщення всіх ресурсів, потрібних для виконання обчислень, описаних в Graph (в Akka Streams це часто буде включати запуск акторів). Дякуючи тому, що Flows просто описує конвеєра обробки, вони є *незмінними, потоко-безпечними, та вільно розподіленими*, що означає, що, наприклад, безпечно поділяти та надсилати їх між акторами, щоб один актор підготував обробку, та потім матеріалізувати його в деякій повністю інішій частині кода.

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)

// приєднуємо Source до Sink, отримуючи RunnableGraph
val runnable: RunnableGraph[Future[Int]] = source.toMat(sink)(Keep.right)

// матеріалізуємо потік, та отримуємо значення FoldSink
val sum: Future[Int] = runnable.run()
```

Після виконання (матеріалізації) `RunnableGraph[T]` ми отримуємо назад матеріалізоване значення типу T. Кожна стадія обробки може продукувати матеріалізоване значення, і це є відповідальністю користувача скомбінувати їх в новий тип. В прикладі вище ми використовували `toMat`, щоб вказати, що ми бажаємо трансформувати матеріалізоване значення джерела та приймача, та ми використали зручну функцію `Keep.right`, щоб сказати, що ми зацікавлені тільки в матеріалізованому значенні приймача. В нашому прикладі `FoldSink` матеріалізує значення типа `Future`, що буде представляти результат процесу згортання потоку. Загалом, потік може показувати декілька матеріалізованих значень, але є досить загальним цікавитись тільки значеннями джерела або приймача потока. З цієї причини є зручний метод з назвою `runWith()`, доступний для `Sink`, `Source` або `Flow`, що вимагає відповідно, наданного `Source` (щоб робити з `Sink`), `Sink` (щоб робити з `Source`), або обох, `Source` та `Sink` (щоб виконувати `Flow`, оскільки жодний ще не під'єднаний).

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)

// матеріалізуємо потік, отримуючи матеріалізоване значення Sinks
val sum: Future[Int] = source.runWith(sink)
```

Треба зауважити, що оскільки стадії обробки є *незмінними*, їх з'єднання повертає нову стадію обробки, замість модифікації існуючого примірника, так що при конструюванні довгих потоків пам'ятайте присвоювати нове значення до змінної, або виконати його:

```
val source = Source(1 to 10)
source.map(_ => 0) // ніякого ефекту на source, оскільки воно незмінне
source.runWith(Sink.fold(0)(_ + _)) // 55

val zeroes = source.map(_ => 0) // повертає нове Source[Int], з доданим `map()`
```

```
6. zeroes.runWith(Sink.fold(0)(_ + _)) // 0
```

**Зауваження**

По замовчанню елементи Akka Streams підтримують **рівно одну** стадію обробки даунстріму. Створення розгалуження (що підтримує декілька стадій обробки даунстріму) є явною можливістю обробки, що дозволяє елементам потоку по замовчанню бути менш складними та більш ефективними. Це також дозволяє більшу гнучкість щодо того, *як точно* обробляти мультикастні сценарії, через провадження іменованих елементів розгалуження, таких, як шикрокомовлення (сигналізує всім елементам даунстріму) або баланс (сигналізує одному з доступних елементів даунстріму).

В прикладі вище ми використовували метод `runWith` , що обоє, матеріалізує потік, та повертає матеріалізоване значення наданого приймача або потоку.

Оскільки потік може бути матеріалізований декілька разів, матеріалізоване значення також буде обчислюватись по новій для кожної такої матеріалізації, що зазвичай призведе до різних значень, що будуть повертатись кожного разу. В прикладі нижче ми створюємо два робочих матеріалізованих примірники потоку, що ми описали в змінній `runnable`, та обоє матеріалізації дають нам різні `Future` з мапи, навіть не зважаючи на те, що ми використовували однаковий `sink` для посилання на футуру:

```
1. // з'єднуємо Source до Sink, отримуючи RunnableGraph
2. val sink = Sink.fold[Int, Int](0)(_ + _)
3. val runnable: RunnableGraph[Future[Int]] =
4.   Source(1 to 10).toMat(sink)(Keep.right)
5.
6. // отримати матеріалізоване значення FoldSink
7. val sum1: Future[Int] = runnable.run()
8. val sum2: Future[Int] = runnable.run()
9.
10. // sum1 та sum2 є різними Future!
```

## Визначення джерел, приймачів та потоків

Об'єкти `Source` та `Sink` визначають різні шляхи для створення джерел та приймачів елементів. наступні приклади показують деякі з найбільш корисних конструкцій (посилайтесь на документацію API щодо додаткових деталей):

```
1. // Створює джерело з Iterable
```

```
Source(List(1, 2, 3))

// Створює джерело з Future
Source.fromFuture(Future.successful("Hello Streams!"))

// Створює джерело з єдиного елемента
Source.single("only one element")

// Порожнє джерело
Source.empty

// Приймач, що згортає потік та повертає Future
// загального результата як матеріалізоване значення
Sink.fold[Int, Int](0)(_ + _)

// Приймач, що повертає Future в якості матеріалізованого значення,
// що містить перший елемент потоку
Sink.head

// Sink, що споживає потік, ні чого не роблячі з елементами
Sink.ignore

// Sink, що виконує виклик-побічний ефект для кожного елемента of the stream
Sink.foreach[String](println(_))
```

Є різноманитні шляхи зв'язати різні частини потоку, наступні приклади показують деякі з доступних опцій:

```
// Явно створити та підключити Source, Sink та Flow
Source(1 to 6).via(Flow[Int].map(_ * 2)).to(Sink.foreach(println(_)))

// Починаємо з Source
val source = Source(1 to 6).map(_ * 2)
source.to(Sink.foreach(println(_)))
```

```
7.
8.  // Починаємо з Sink
9.  val sink: Sink[Int, NotUsed] = Flow[Int].map(_ * 2).to(Sink.foreach(println(_)))
10. Source(1 to 6).to(sink)
11.
12. // Широкомовлення до приймачів один за одним
13. val otherSink: Sink[Int, NotUsed] =
14.   Flow[Int].alsoTo(Sink.foreach(println(_))).to(Sink.ignore)
15. Source(1 to 6).to(otherSink)
```

### Нелегальні елементи потоку

У відповідності зі специфікацією Reactive Streams (Правило 2.13) Akka Streams не дозволяють `null` бути переданим через потік в якості елемента. В випадку, коли ви бажаєте моделювати концепцію відсутності значення, ми рекомендуємо використовувати `scala.Option` або `scala.util.Either`.

## Пояснення зворотнього тиску

Akka Streams реалізує асинхронний неблокуючий протокол зворотнього тиску, стандартизований в специфікації Reactive Streams, до якого Akka є членом-засновником.

Користувач бібліотеки не має писати любий явний код обробки зворотнього тиску — він вбудований, та вирішує справи автоматично, під час всіх стадій обробки, запроваджених Akka Streams. Однак можливо додавати явні стадії буферізації зі стратегіями переповнення, що можуть впливати на поведінку потока. Це особливо важливо в комплексних графах обробки, що навіть можуть містити цикли (що *мусять* розглядатись з дуже прискіпливою увагою, як пояснено в Циклічних графах, дієздатність та глухе блокування).

Керування зворотнім тиском визначений в термінах числа елементів в даунстримі, що `Subscriber` в змозі отримати та буферизувати, на який посилаються, як на `demand`. Джерело даних, відоме як `Publisher` в термінології Reactive Streams, та реалізоване як `Source` в Akka Streams, гарантує, що воно ніколи не буде видавати більше елементів, ніж отримана загальна протреба для даного `Subscriber`.

**Зауваження**

Специфікація Reactive Streams визначає свій протокол в термінах `Publisher` та `Subscriber`. Ці типи **не призначені** бути частиною користувацького API, та замість цього вони служать як низькорівневі будівельні блоки для різних реалізацій Reactive Streams.

Akka Streams реалізує ці концепції як `Source`, `Flow` (відомий як `Processor` в Reactive Streams), та `Sink`, без розкриття інтерфейсів Reactive Streams напряму. Якщо вам треба інтегруватись з іншими бібліотеками Reactive Stream, читайте Інтеграція з Reactive Streams.

Режим, в якому робить зворотній тиск в Reactive Streams може бути влучно описаний як "динамічний режим push/pull", оскільки він перемикається між push та pull на основі моделей зворотнього тиску, в залежності на даунстрімі, та чи може він впоратись з виробництвом апстріму, чи ні.

Щоб проіллюструвати це більш докладно, давайте розглянемо обидві проблемні ситуації, та як протокол зворотнього тиску обробляє їх:

## Повільний Publisher, швидкий Subscriber

Це, звичайно, щасливий випадок – нам не треба уповільнювати Publisher в цьому випадку. Однак сигнальна частота рідко стала, та може змінитись в кожну мить, несподівано закінчившись в ситуації, коли Subscriber тепер повільніший, ніж Publisher. Для того, щоб вберегтись від цієї читуації, протокол зворотнього тиску має все ще бути ввімкненим на протязі таких ситуацій, однак ми не бажаємо платити щирі пенальті за те, що цей запобіжник ввімкнено.

Протокол Reactive Streams вирішує це через асинхронну сигналізацію від Subscriber до Publisher сигналами `Request(n:Int)`. Протокол гарантує, що Publisher ніколи не надішле *більше елементів,* ніж вказано в запиті. Однак оскільки Subscriber наразі швидший, він буде сигналити ці повідомлення-запити з вищою швидкістю (та можливо також пакуючи запити разом - запитиючи декілька елементів в одному сигналі). Це означає, що Publisher не повинен будь-коли простоювати (бути під зворотнім тиском) під час надсилання своїх входячих елементів.

Як ви можете бачити, в цьому сценарії ми ефективно оперуємо в так званому push-режимі, оскільки Publisher може продовжувати продукувати елементи так швидко, як може, оскільки підвислий запит буде вчасно визначений під час видачі елементів.

## Швидкий Publisher, повільний Subscriber

Це той випадок, коли зворотній тиск на `Publisher` необхідний, оскільки `Subscriber` не в змозі впоратись зі швидкістю, з якою його апстрим буде видавати дані.

Оскільки `Publisher` не дозволено сигналити більше елементів, тому завислі запити, просигналені `Subscriber`, він буде дотримуватись цього зворотнього тиску через застосування однієї з наступних стратегій:

- не генерувати елементи, якщо це дозволить контролювати їх частоту продукування,
- через буферизацію елементів в *обмеженій манері*, доки не буде просигналений більший запит,
- відкидати елементи, доки не буде просигналений більший запит,
- зривати потік, якщо неможливо застосувати жодну з стратегій вище.

Як ми можемо бачити, цей сценарій ефективно означає, що `Subscriber` буде *підтягувати (pull)* елементи від Publisher – на цей режим операцій посилаються як на pull-базований зворотній тиск.

# Матеріалізація потоків

При конструюванні потоків та графів в Akka Streams думайте про них як про підготовку схеми, плана виконання. Матеріалізація потока є процесом прийняття опису потока (графа), та розташування всіх потрібних ресурсів, що він потребує для того, щоб робити. В випадку Akka Streams це часто означає запуск акторів, що виконують обробку, але не обмежується цім — це також може означати відкриття файлів або сокетних з'єднань, таке інше — в залежності від того, що потребує потік.

Матеріалізація вмикається на так званих "термінальних операціях". Більш конкретно, це включає різні форми методів `run()` та `runWith()`, визначені на елементах `Source` та `Flow`, а також як невелика кількість специфічного синтаксичного цукру для добре-знаних приймачах, такі, як `runForeach(el => ...)` (що є псевдонимом до `runWith(Sink.foreach(el => ...))`.

Матеріалізація наразі виконується синхронно на потоці матеріалізації. Справжня обробка потоків виконується акторами, що запускаються під час матеріалізації потоків, та будуть виконуватись на пулі потоків, на якому вони сконфігуровані виконуватись - що, по замовчанню, є налаштування диспечерів в `MaterializationSettings` під час конструювання `ActorMaterializer`.

**Зауваження**

Повторне використання *примірників стадій* лінійних обчислень (Source, Sink, Flow) в композитних графах є легальним, що буде матеріалізувати їх декілька разів.

## Сплав операторів

Akka Streams 2.0 містить початкову версію підтримки сплаву потокових операцій. Це означає, що кроки обробки потоку або графу можуть бути виконані на тому ж акторі, та має три наслідки:

- запуск потоку може зайняти довше, ніж раніше, через виконання алгоритму сплавлення
- передання елементів від однієї до стадії до наступної є більш швидким між сплавленими стадіями, через уникання накладних розходів асинхронних повідомлень
- сплавлені процеси стадій обробки більше роблять паралельно один до одного, що означає, що для кожної частини використовується тільки одне ядро CPU

Перша точка, що може бути знайдена пре-сплавкою, та потім повторно використана, є схема потоку, що змальована нижче:

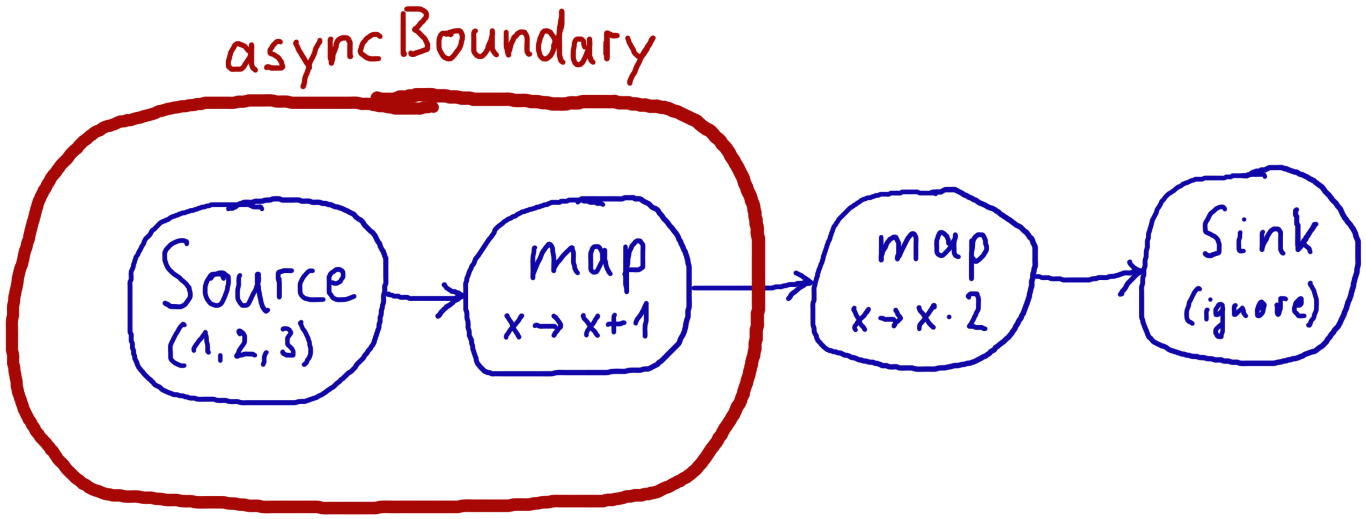
```
1. import akka.stream.Fusing
2.
3. val flow = Flow[Int].map(_ * 2).filter(_ > 500)
4. val fused = Fusing.aggressive(flow)
```

```
5.
6. Source.fromIterator { () => Iterator from 0 }
7.    .via(fused)
8.    .take(1000)
```

Щоб збалансувати ефекти другої та третьої точок, ви маєте вручну вставити асинхронні обмеження в ваші потоки та графи, шляхом додавання `Attributes.asyncBoundary` з використанням методу `async` на `Source`, `Sink` та `Flow` на частини, що буде комунікувати з залишком графа в асинхронній манері.

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

В цьому прикладі ми створюємо два регіони в потоці, що будуть виконуватись в одному акторі кожний — вважаючи, що додавання та множення цілих є вкрай коштовною операцією, це буде призводити до приросту продуктивності, оскільки два CPU можуть обробляти ці задачі паралельно. Є важливим зауважити, що асинхронні межі не є місцями сингулярності в потоці, де елементи передаються асинхронно (як в інших поточних бібліотеках), але, замість цього, атрибути завжди роблять через додавання інформації до графу потока, що був побудований до цієї точки:



Це означає, що будь що, що є всередині червоної бульбашки, буде виконане одним актором, а все інше - іншим. Ця схема може бути успішно застосована, завжди маючи таке обмеження попередніх, плюс всіх стадій обробки, що були додані після них.

**Попередження**

Без сплавки (тобто, до версії 2.0-M2), кожний потік стадії обробки мав неявний вхідний буфер, що містить декілька елементів з причин ефективності. Якщо ваш граф потоку містить цикли, тоді ці буфери можуть бути критичними, щоб запобігти глухому блокуванню. Зі сплавкою цих неявних буферів вже немає, елементи даних передаються без буферизації між сплавленими стадіями. В таких випадках, де буферізація потрібна щоб взагалі забезпечити роботу потока, ви можете задати вставку явних буферів за допомогою комбінатора `.buffer()` — типово буфера розміру 2 достатньо, щоб дозволити зворотній зв'язок до функції.

Нова поведінка сплавки може бути відключена через встановлення параметра конфігурації `akka.stream.materializer.auto-fusing=off`. В такому випадку ви все ще можете вручну сплавити ті графи, що будуть робити на меньшій кількості акторів. За винятком `SslTlsStage` та оператора `groupBy`, всі вбудовані стадії обробки можуть бути сплавлені.

## Комбінація матеріалізованих значень

Оскількьки кожна стадія обробки в Akka Streams може провадити матеріалізоване значення, потрібно якось виражати, як ці значення мають бути скомпоновані в заключне значення, коли ці стадії підключаються разом. Для цього багато методів-комбінаторів мають варіанти, що приймають додатковий аргумент, функцію, що буде використана для комбінації отриманих значень. Деякі приклади використання цих комбінаторів проілюстровані в прикладі нижче.

```
1. // Джерело, що можна просигналити явно ззовні
2. val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]
3.
4. // Потік, що зсередини призупиняє елементи до одного на секунду, та повертає Cancellable,
5. // що може бути використане для завершення потоку
6. val flow: Flow[Int, Int, Cancellable] = throttler
7.
8. // Приймач, що повертає перший елемент потоку в повернутій Future
9. val sink: Sink[Int, Future[Int]] = Sink.head[Int]
10.
11. // По замовчанню зберігається матеріалізоване значення самої лівої стадії
12. val r1: RunnableGraph[Promise[Option[Int]]] = source.via(flow).to(sink)
```

```
13.
14. // Простий вибір матеріалізованих значень через використання Keep.right
15. val r2: RunnableGraph[Cancellable] = source.viaMat(flow)(Keep.right).to(sink)
16. val r3: RunnableGraph[Future[Int]] = source.via(flow).toMat(sink)(Keep.right)
17.
18. // Використовуючи runWith завжди буде давати мат.значення стадій, доданих самим runWith()
19. val r4: Future[Int] = source.via(flow).runWith(sink)
20. val r5: Promise[Option[Int]] = flow.to(sink).runWith(source)
21. val r6: (Promise[Option[Int]], Future[Int]) = flow.runWith(source, sink)
22.
23. // Використання більш складних комбінацій
24. val r7: RunnableGraph[(Promise[Option[Int]], Cancellable)] =
25.   source.viaMat(flow)(Keep.both).to(sink)
26.
27. val r8: RunnableGraph[(Promise[Option[Int]], Future[Int])] =
28.   source.via(flow).toMat(sink)(Keep.both)
29.
30. val r9: RunnableGraph[((Promise[Option[Int]], Cancellable), Future[Int])] =
31.   source.viaMat(flow)(Keep.both).toMat(sink)(Keep.both)
32.
33. val r10: RunnableGraph[(Cancellable, Future[Int])] =
34.   source.viaMat(flow)(Keep.right).toMat(sink)(Keep.both)
35.
36. // Також можливо відзеркалити матеріалізовані значення. В r9 ми маємо двічі
37. // вкладену пару, але ми бажаємо спрямити її
38. val r11: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
39.   r9.mapMaterializedValue {
40.     case ((promise, cancellable), future) =>
41.       (promise, cancellable, future)
42.   }
43.
44. // Тепер ми можемо використати рівняння, щоб виділити отримані матеріалізовані значення
45. val (promise, cancellable, future) = r11.run()
```

```
46.
47. // Виведення типів робить як звичайно
48. promise.success(None)
49. cancellable.cancel()
50. future.map(_ + 3)
51.
52. // Результат r11 може також бути досягнений через використання Graph API
53. val r12: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
54.   RunnableGraph.fromGraph(GraphDSL.create(source, flow, sink)((_, _, _)) { implicit builder => (src, f, dst) =>
55.     import GraphDSL.Implicits._
56.     src ~> f ~> dst
57.     ClosedShape
58.   })
```

**Зауваження**

В графах можливо отримати доступ до матеріалізованого значення з середини обробки потоку графа. Щодо деталей дивіться Доступ до матеріалізованого значення в Graph.

## Впорядкування потоків

В Akka Streams майже всі стадії обчислення *зберігають вхідний порядок* елементів. Це означає, що якщо на вхід надійде `{IA1,IA2,...,IAn}`, це "спричинить" вихід `{OA1,OA2,...,OAk}`, та на вході `{IB1,IB2,...,IBm}` "спричинить" на виході `{OB1,OB2,...,OBl}`, та якщо всі `IAi` трапляться перед всіма `IBi`, тоді `OAi` трапиться перед `OBi`.

Ця властивість дотримується навіть для асинхронних операцій, таких, як `mapAsync`, однак існує невпорядкована версія, зо називаєтьсья`mapAsyncUnordered`, що не зберігає цей порядок.

Однак в випадку поєднань, що обробляють декілька вхідних потоків (тобто `Merge`), вихідний порядок, загалом, *не визначений* для елементів, що з'являються на різних вхідних портах. Таким чином, операції як злиття можуть видавати `Ai` перед `Bi`, та це залежить від внутрішньої логіки, як визначати порядок видачі елементів. Спеціалізовані елементи, такі як `Zip`, однак, *гарантують* власний порядок виходу, тому що кожний вихідний елемент залежить від всіх елементів апстриму, що вони вже просигналені – таким чином, порядок в випадку поєднання zipping визначений цією властивістю.

Якщо ви винайдете, що вам потрібно гарний контроль за порядком виданих елементів в сценаріях поєднання потоків, розгляньте використання `MergePreferred` або `GraphStage` – що дає вам повний контроль над тим, як виконувати злиття.

# Робота з графами

В Akka Streams графи обчислень не виражені з використанням поточного DSL, як це є з обчисленнями, замість цього, вони пишуться в більш графо-подібному DSL, що націлений на транслювання з малюнків графа (тобто, з нотатків, що беруться з дискусії по дизайну, або ілюстрацій в специфікації протокола), в код та з кода, простішим. В цьому розділі ми поглинемо в різні шляхи конструювання та використання графів, так само, як пояснимо загальні пастки, та як їх уникати.

Графи потрібні кожного разу, коли ви бажаєте виконати любий різновид операцій злиття ("багато входів") або розгалужень ("багато виходів"). Можна розглядати  Flows як дорогу, ми можемо намалювати операції графа, як операції перетинів: багато потоків поєднуються в одній точці. Деякі операції графів, що є досить загальними, та підходять під лінійний стиль Flows, такі як `concat` (що конкатенує два потоки, так що другий споживається після того, як завершений перший), може бути скороченим методом, визначеним на самих `Flow` або `Source`, однак ви повинні мати на увазі, що вони також реалізовані як поєднання графів.

## Конструювання графів

Графи будуються з простих Flows, що служать як лінійні з'єднання в графі, а  також поєднання, що служать як точки розгалуження та поєднання для Flows. Дякуючи поєднанням, що мають осмислені типи, базуючись на їх поведінці, та роблячи їх явними елементами, ці елементи стає просто використовувати.

Akka Streams наразі провадить наступні поєднання (для детального списку дивіться Огляд вбудованих стадій та їх семантика):

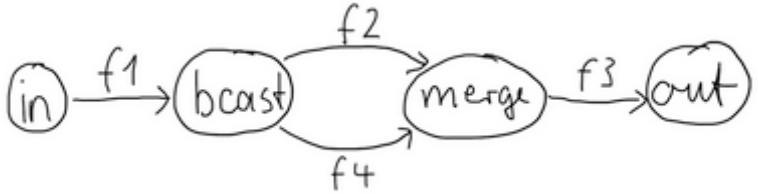
- **Розгалуження**

  - `Broadcast[T]` – *(1 вхід, N виходів)* призводить до того, що кожний елемент на вході подається на всі виходи
  - `Balance[T]` – *(1 вхід, N виходів)* елементи на вході подаються на один з декількох вихідний порт
  - `UnzipWith[In,A,B,...]` – *(1 вхід, N виходів)* приймає функцію з одного входа, що надає значення на N вихідних елементів (де N <= 20)
  - `UnZip[A,B]` – *(1 вхід, 2 виходи)* ділить кортежі `(A,B)` на два потоки, один типу `A`, та один типу `B`

- **Поєднання**

  - `Merge[In]` – *(N входів, 1 вихід)* приймає з входів в довільному порядку, та надсилає по одному на вихід
  - `MergePreferred[In]` – як `Merge`, але якщо доступне на порті `preferred`, приймає з нього, інакше довільно з `others`
  - `ZipWith[A,B,...,Out]` – *(N входів, 1 вихід)* приймає функцію з N входів, що приймаючи з кожного входу видає один вихідний елемент
  - `Zip[A,B]` – *(2 входи, 1 вихід)* є варіантом `ZipWith`, що спеціалізується на поєднанні двох вхідних потоків `A` та `B` в потік кортежів `(A,B)`
  - `Concat[A]` – *(2 входи, 1 вихід)* конкатенує два потоки (споживає спочатку перший, потім другий)

Однією з цілей GraphDSL DSL є виглядати подібно до того, як дехто може намалювати граф на дошці, так що стає простішим транслювати дизайн з дошки на код, та бути в змозі поєднати один з одним. Давайте проілюструємо це через трансляцію намальованого нижче графа в Akka Streams:




Такий граф просто транслюється на Graph DSL, оскільки кожний лінійний елемент відповідає до `Flow`, та кожне коло відповідає або до `Junction`, або `Source`, або `Sink`, якщо воно починає або закінчує `Flow`. Поєднання завжди мають бути створені з визначеними параметрами типу, або інакше буде виведений тип `Nothing`.

```
1.  val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder: GraphDSL.Builder[NotUsed] =>
2.    import GraphDSL.Implicits._
3.    val in = Source(1 to 10)
4.    val out = Sink.ignore
5.
6.    val bcast = builder.add(Broadcast[Int](2))
7.    val merge = builder.add(Merge[Int](2))
8.
9.    val f1, f2, f3, f4 = Flow[Int].map(_ + 10)
10.
11.   in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
12.   bcast ~> f4 ~> merge
13.   ClosedShape
14. })
```

**Зауваження**

Визначення *референтої еквівалентності* поєднань визначає *еквівалентність вузлів графа* (тобто, той самий *примірник* злиття, використаний в GraphDSL, посилається на те ж саме розташування в результуючому графі).

Зауважте `import GraphDSL.Implicits._`, що привносить до поля зору оператор `~>` (читається як "вістря", "через" або "до"), та зворотній до нього `<~` (щоб нотувати потоки в зворотньому напрямку, коли це потрібно).

Дивлячись на код вище, має бути зрозумілим, що об'єкт `GraphDSL.Builder` є *змінним*. Він (неявно) використовує оператором `~>`, також роблячи це змінною операцією. Причина вибору такого дизайну є дозволити простіше створення складних графів, що навіть можуть містити цикли. Але, однак, коли GraphDSL був сконструйований, примірник `GraphDSL` *є незмінним, потоко-безпечним, та вільно розповсюджується*. Те ж саме вірно для всіх частин графа — джерел, приймачів, та потоків — коли вони сконструйовані. Це означає, що ви можете безпечно повторно виконувати отриманий Flow або поєднання в багатьох місцях в оброблюваному графі.

Ви бачили приклади такого використання вище: поєднання злиття та розгалуження були імпортовані в граф з використанням `builder.add(...)`, та операція, що буде робити копію схеми, що ви передали їй, та повертати входи та виходи отриманої копії, так, щоб вони могли бути поєднані. Інша альтернатива полягає в передачі існуючих графів — любої форми — в метод-фабрику, що продукує новий граф. Різниця між ціма підходами в тому, що імпортування з використанням `builder.add(...)` ігнорує матеріалізоване значення імпортованого графа, тоді як імпортування через метод-фабрику дозволяє його включення; щодо подробиць дивіться Матеріалізація потоків.

В прикладі нижче ми готуємо граф, що складається з двох паралельних потоків, в обох з яких ми використовуємо той же примірник `Flow`, але це також буде відповідно матеріалізовано як два з'єднання між відповідними Sources та Sinks:

```
1.  val topHeadSink = Sink.head[Int]
2.  val bottomHeadSink = Sink.head[Int]
3.  val sharedDoubler = Flow[Int].map(_ * 2)
4.
5.  RunnableGraph.fromGraph(GraphDSL.create(topHeadSink, bottomHeadSink)((_, _)) { implicit builder =>
6.    (topHS, bottomHS) =>
7.    import GraphDSL.Implicits._
8.    val broadcast = builder.add(Broadcast[Int](2))
9.    Source.single(1) ~> broadcast.in
10.
11.   broadcast.out(0) ~> sharedDoubler ~> topHS.in
12.   broadcast.out(1) ~> sharedDoubler ~> bottomHS.in
13.   ClosedShape
14. })
```

# Конструювання та комбінація часткових графів

Іноді неможливо (або не треба) сконструювати цілий граф обчислень в одному місці, але замість цього сконструювати всі різні фази в різних місцях, та в кінці поєднати їх в завершений граф, та виконати його.

Цього можливо досягти, повертаючи інший `Shape` ніж `ClosedShape`, наприклад, `FlowShape(in, out)`, з функції, наданої до `GraphDSL.create`. Дивіться Попередньо визначені форми для списка попередньо визначених форм.

Перетворення `Graph` на `RunnableGraph` потребує під'єднання всіх портів, та якщо цього не зробити, буде викликане виключення під час конструювання, що допомагає уникнути простих помилок під час роботи з графами. Частковий граф, однак, дозволяє вам повертати набір ще призначених для під'єднання портів з блока кода, що виконує внутрішнє підключення.

Давайте уявимо, що ми бажаємо провадити користувачів спеціалізованим елементом, що приймає три вхідні елементи, та обирає найбільше ціле з кожної об'єднаної пари. Ми бажаємо показати три вхідні порти (непід'єднані джерела), та один вихідний порт (непід'єднаний приймач).

```
1. val pickMaxOfThree = GraphDSL.create() { implicit b =>
2.   import GraphDSL.Implicits._
3.
4.   val zip1 = b.add(ZipWith[Int, Int, Int](math.max _))
5.   val zip2 = b.add(ZipWith[Int, Int, Int](math.max _))
6.   zip1.out ~> zip2.in0
7.
8.   UniformFanInShape(zip2.out, zip1.in0, zip1.in1, zip2.in1)
9. }
10.
11. val resultSink = Sink.head[Int]
12.
13. val g = RunnableGraph.fromGraph(GraphDSL.create(resultSink) { implicit b => sink =>
14.   import GraphDSL.Implicits._
15.
16. // імпортування часткового графа буде повертати його форму (входи та виходи)
17.   val pm3 = b.add(pickMaxOfThree)
18.
19.   Source.single(1) ~> pm3.in(0)
```

```
20.   Source.single(2) ~> pm3.in(1)
21.   Source.single(3) ~> pm3.in(2)
22.   pm3.out ~> sink.in
23.   ClosedShape
24. })
25.
26. val max: Future[Int] = g.run()
27. Await.result(max, 300.millis) should equal(3)
```

Як ви можете бачити, спершу ми конструюємо частковий граф, що містить всі поєднання, та порівнює елементи потоку. Цей частковий граф буде мати три входи, та один вихід, таким чином ми використовуємо `UniformFanInShape`. Коли ми явно імпортуємо його (всі входи та з'єднання) в закритий граф, що ми будуємо на другому кроці, де всі невизначені елементи підключаються до реальних джерел та приймачів. Потім граф може виконатись та отримати очікуваний результат.

**Попередження**

Будь ласка, занотуйте, що `GraphDSL` не в змозі провадити безпечність типів часу компіляції, щодо того, чи всі елементи були вірно під'єднані — ця перевірка виконується під час виконання під час втілення графа.

Частковий граф також перевіряє, що всі порти або під'єднані, або частка повертає `Shape`.

# Побудова джерел, приймачів та потоків з часткових графів

Замість того, щоб трактувати частковий граф як просто колекцію потоків та з'єднань, що ще можуть бути непід'єднаними, часом корисно показати такі складні графи як простіші структури, як `Source`, `Sink` або `Flow`.

Фактично, ці концепції можуть бути просто виражені як спеціальні випадки частково поєднаних графів:

- `Source` є частковий граф з *точно одним виходом*, що він повертає як `SourceShape`.
- `Sink` є частковим графом, з *точно одним входом*, що він повертає як `SinkShape`.
- `Flow` є частковим графом з *точно одним входом та одним виходом*, що повертає `FlowShape`.

Маючи змогу приховати складні графи в простих елементах, таких як Sink/Source/Flow дозволяє вам просто створювати один великий елемент, та з цього часу розглядати його як просту составну стадію для лінійних обчислень.

Щоб створити Source з графа використовується метод `Source.fromGraph`, та щоб застосувати його ми повинні мати `Graph[SourceShape, T]`. Він конструюється `GraphDSL.create` та повертає `SourceShape` з переданої йому функції. Поодинокий вихід має провадити до метода `SourceShape.of`, та стає елементом, що потребує приймач, щоб стати спроможним до виконання.

Посилайтесь на приклад нижче, в якому ми створюємо Source, що поєднує разом два числа, щоб побачити цю конструкцію графа в дії:

```
1. val pairs = Source.fromGraph(GraphDSL.create() { implicit b =>
2.   import GraphDSL.Implicits._
3.
4.   // готуємо елементи графа
5.   val zip = b.add(Zip[Int, Int]())
6.   def ints = Source.fromIterator(() => Iterator.from(1))
7.
8.   // з'єднуємо граф
9.   ints.filter(_ % 2 != 0) ~> zip.in0
10.  ints.filter(_ % 2 == 0) ~> zip.in1
11.
12.  // показуємо порт
13.  SourceShape(zip.out)
14. })
15.
16. val firstPair: Future[(Int, Int)] = pairs.runWith(Sink.head)
```

Приблизно те саме може бути зроблено для `Sink[T]`, використовуючи `SinkShape.of`, в якому випадку запроваджене значення `Inlet[T]`. Для визначення `Flow[T]` нам треба показати обоє, вхідне та вихідне під'єднання:

```
1. val pairUpWithToString =
2.   Flow.fromGraph(GraphDSL.create() { implicit b =>
3.     import GraphDSL.Implicits._
4.
5.     // готуємо елементи графа
6.     val broadcast = b.add(Broadcast[Int](2))
7.     val zip = b.add(Zip[Int, String]())
```

```
8.
9.      // з'єднуємо граф
10.     broadcast.out(0).map(identity) ~> zip.in0
11.     broadcast.out(1).map(_.toString) ~> zip.in1
12.
13.     // показуємо порти
14.     FlowShape(broadcast.in, zip.out)
15.   })
16.
17. pairUpWithToString.runWith(Source(List(1)), Sink.head)
```

## Комбінування джерел та приймачів за допомогою спрощеного API

Існує спрощений API, що ви можете застосовувати для комбінування джерел з поєднаннями, такі як: `Broadcast[T]`, `Balance[T]`, `Merge[In]` та `Concat[A]`, без потреби використовувати Graph DSL. Метод combine турбується про конструювання відповідного графа за лаштунками. В наступному прикладі ми скомбінуємо два джерела в одне  (поєднання fan-in):

```
1. val sourceOne = Source(List(1))
2. val sourceTwo = Source(List(2))
3. val merged = Source.combine(sourceOne, sourceTwo)(Merge(_))
4.
5. val mergedResult: Future[Int] = merged.runWith(Sink.fold(0)(_ + _))
```

Те ж саме може бути зроблене з `Sink[T]`, але цього разу для розгалуження:

```
1. val sendRmotely = Sink.actorRef(actorRef, "Done")
2. val localProcessing = Sink.foreach[Int](_ => /* робимо щось корисне */ ())
3.
4. val sink = Sink.combine(sendRmotely, localProcessing)(Broadcast[Int](_))
5.
6. Source(List(0, 1, 2)).runWith(sink)
```

# Побудова повторно використовуваних компонентів графа

Можливо побудувати повторно використовувані, інкапсульовані компоненти з довільними вхідними та вихідними портами з використанням графового DSL.

Як приклад, ми будемо будувати з'єднання графів, що представляє пул робітників, де робітник виражений як `Flow[I,O,_]`, тобто проста трансформація завдань типу `I` в тип `O` (як ви вже бачили, цей потік може мати зсередини складний граф). Наш повторно використовуваний пул робітників не буде зберігати порядок надходячих завдань (вважатимемо, що вони мають відповідне поле ID), та буде використовувати поєднання `Balance` для планування завдань до доступних робітників. Зверху цього наше поєднання буде мати "швидкісну полосу", виділений порт, по якому можуть надсилатись завдання з підвищенним приоритетом.

Загалом, наше поєднання буде мати два вхідних порти типу `I` (для нормальних та приоритизованих завдань), та вихід типу `O`. Щоб представити цей інтерфейс, Щоб представити цей інтерфейс, нам треба визначити власний `Shape`. Наступні рядки показують як зробити це.

```
1. // Форма, що представляє вхідні та вихідні порти повторно використовуваного модуля
2. case class PriorityWorkerPoolShape[In, Out](
3.   jobsIn:         Inlet[In],
4.   priorityJobsIn: Inlet[In],
5.   resultsOut:     Outlet[Out]) extends Shape {
6.
7.   // Є важливим провадити список всіх вхідних та вихідних портів з
8.   // стабільним порядком. Дублікати не допустимі.
9.   override val inlets: immutable.Seq[Inlet[_]] =
10.    jobsIn :: priorityJobsIn :: Nil
11.  override val outlets: immutable.Seq[Outlet[_]] =
12.    resultsOut :: Nil
13.
14.  // Shape має вміти створювати свою копію. Загалом це означає новий примірник з копіями портів
15.  override def deepCopy() = PriorityWorkerPoolShape(
16.    jobsIn.carbonCopy(),
17.    priorityJobsIn.carbonCopy(),
18.    resultsOut.carbonCopy())
19.
20.  // Shape має також бути в змозі створювати себе з існуючих портів
```

```
21.   override def copyFromPorts(
22.     inlets:  immutable.Seq[Inlet[_]],
23.     outlets: immutable.Seq[Outlet[_]]) = {
24.     assert(inlets.size == this.inlets.size)
25.     assert(outlets.size == this.outlets.size)
26.     // Ось чому порядок має значення при перевизначенні входів та виходів.
27.     PriorityWorkerPoolShape[In, Out](inlets(0).as[In], inlets(1).as[In], outlets(0).as[Out])
28.   }
29. }
```

## Попередньо визначені фігури

Загалом власний `Shape` має бути в змозі запровадити всі свої вхідні та вихідні порти, бути в змозі копіювати себе, а також вміти створювати себе з наданих портів. Існують деякі попередньо визначені порти, надані для уникання непотрібному повторюванню:

- `SourceShape`, `SinkShape`, `FlowShape` для простіших фігур,
- `UniformFanInShape` та `UniformFanOutShape` для поєднань з багатьма вхідними (або вихідними) портами того ж самого типу,
- `FanInShape1`, `FanInShape2`, ..., `FanOutShape1`, `FanOutShape2`, ... для поєднань з багатьма вхідними (або вихідними) портами різних типів.

Оскільки наша фігура має два вхідні порти та один вихідний порт, ми можемо просто використати `FanInShape` DSL для визначення нашої власної фігури:

```
1. import FanInShape.{ Init, Name }
2.
3. class PriorityWorkerPoolShape2[In, Out](_init: Init[Out] = Name("PriorityWorkerPool"))
4.   extends FanInShape[Out](_init) {
5.   protected override def construct(i: Init[Out]) = new PriorityWorkerPoolShape2(i)
6.
7.   val jobsIn = newInlet[In]("jobsIn")
8.   val priorityJobsIn = newInlet[In]("priorityJobsIn")
9.   // Outlet[Out] на ім'я "out" створюється автоматично
10. }
```

Тепер ми маємо `Shape`, що ми можемо під'єднати в Graph, що представляє наш пул робітників. Перше, ми будемо зливати вхідні завдання зі звичайним та підвищеним приоритетом, з використанням `MergePreferred`, потім ми надсилатимемо завдання до поєднання `Balance`, що буде розгалужуватись до конфігурованого числа робітників (потоків), та нарешті ми зливаємо всі ці результати разом, та надсилаємо їх через наш єдиний вихідний порт. Це виражене через наступний граф:

```
1.  object PriorityWorkerPool {
2.    def apply[In, Out](
3.      worker:      Flow[In, Out, Any],
4.      workerCount: Int): Graph[PriorityWorkerPoolShape[In, Out], NotUsed] = {
5.
6.      GraphDSL.create() { implicit b =>
7.        import GraphDSL.Implicits._
8.
9.        val priorityMerge = b.add(MergePreferred[In](1))
10.       val balance = b.add(Balance[In](workerCount))
11.       val resultsMerge = b.add(Merge[Out](workerCount))
12.
13.       // Після злиття приоритетних та звичайних завдань, ми направляємо їх на балансування
14.       priorityMerge ~> balance
15.
16.       // Підключаємо кожний з виходів балансувальника до робочого потоку, та потім знову поєднуємо
17. ,     for (i <- 0 until workerCount)
18.         balance.out(i) ~> worker ~> resultsMerge.in(i)
19.
20.       // Тепер ми показуємо вхідні порти priorityMerge, та вихід resultsMerge як наші порти PriorityWorkerPool
21.       // -- все мило огорнуте в наш специфічний Shape
22.       PriorityWorkerPoolShape(
23.         jobsIn = priorityMerge.in(0),
24.         priorityJobsIn = priorityMerge.preferred,
25.         resultsOut = resultsMerge.out)
26.     }
27.
28.   }
29.
```

```
30. }
```

Все що нам треба зробити, це використати наше власне поєднання в графі. Наступний код симулює роботу деяких простих робітниуів та завдання, з використанням простих рядків та роздруковку результатів. Насправді ми використовуємо *два примірники* нашого робітника в нашому пулі-поєднанні, використовуючи `add()` двічі.

```
1. val worker1 = Flow[String].map("step 1 " + _)
2. val worker2 = Flow[String].map("step 2 " + _)
3.
4. RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
5.   import GraphDSL.Implicits._
6.
7.   val priorityPool1 = b.add(PriorityWorkerPool(worker1, 4))
8.   val priorityPool2 = b.add(PriorityWorkerPool(worker2, 2))
9.
10.  Source(1 to 100).map("job: " + _) ~> priorityPool1.jobsIn
11.  Source(1 to 100).map("priority job: " + _) ~> priorityPool1.priorityJobsIn
12.
13.  priorityPool1.resultsOut ~> priorityPool2.jobsIn
14.  Source(1 to 100).map("one-step, priority " + _) ~> priorityPool2.priorityJobsIn
15.
16.  priorityPool2.resultsOut ~> Sink.foreach(println)
17.  ClosedShape
18. }).run()
```

## Двонаправлені потоки

Топологія графів, що буває часто корисним, це коли два потоки надходять в двох напрямках. Візміть для приклада стадію кодека, що серіалізує вихідні повідомлення, та десеріалізує надходячі потокі октетів. Інша така стадія може додавати протокол фрагментування, що додає заголовок довжини до виходячих даних, та розбиває надходячі фрейми знову на оригінальні шматки потоку октетів. Ці дві стадії можуть бути скомпоновані, застосовані одна зверху другої, як частина стеку протоколу. Для ціх цілей існує специальний тип `BidiFlow`, що є графом, що має саме два вхідних з'єднання, та два вихідні з'єжнання. Відповідна фігура називається `BidiShape`, та визначена таким чином:

```
/**
 * Двонаправлений потік елементів, що відповідно має два входи та два виходи:
 *
 * {{{
 *         +------+
 *  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 {
  // опущені деталі реалізації ...
}
```

Двонаправлений потік визначений так само, як і однонаправлений `Flow`, як демонструє зазначений вище кодек:

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

def toBytes(msg: Message): ByteString = {
  // опущені деталі реалізації ...
}

def fromBytes(bytes: ByteString): Message = {
  // опущені деталі реалізації ...
}
```

```
12.
13. val codecVerbose = BidiFlow.fromGraph(GraphDSL.create() { b =>
14.   // конструюємо та додаємо вищий потік, що іде на вихід
15.   val outbound = b.add(Flow[Message].map(toBytes))
16.   // конструюємо та додаємо нижній потік, що іде на вхід
17.   val inbound = b.add(Flow[ByteString].map(fromBytes))
18.   // зплавляємо їх разом в BidiShape
19.   BidiShape.fromFlows(outbound, inbound)
20. })
21.
22. // це те ж саме, що і раніше
23. val codec = BidiFlow.fromFunctions(toBytes _, fromBytes _)
```

Перша версія повторює конструктор часткового графа, тоді як для простого випадка функціонала перетворення 1:1 є стисліший зручний метод, як показано в останньому рядку. Реалізація двох функцій також не складна:

```
1. def toBytes(msg: Message): ByteString = {
2.   implicit val order = ByteOrder.LITTLE_ENDIAN
3.   msg match {
4.     case Ping(id) => ByteString.newBuilder.putByte(1).putInt(id).result()
5.     case Pong(id) => ByteString.newBuilder.putByte(2).putInt(id).result()
6.   }
7. }
8.
9. def fromBytes(bytes: ByteString): Message = {
10.   implicit val order = ByteOrder.LITTLE_ENDIAN
11.   val it = bytes.iterator
12.   it.getByte match {
13.     case 1     => Ping(it.getInt)
14.     case 2     => Pong(it.getInt)
15.     case other => throw new RuntimeException(s"parse error: expected 1|2 got $other")
16.   }
17. }
```

В такий спосіб ви можете просто інтегрувати любі інші бібліотеки серіалізації, що перетворюють об'єкт на потік байт.

Інша стадія, про яку ми казали, трохи більш складніша, оскільки реверс протокола фрагментування означає, що кожний отриманий пакет байт може відповідати нулю або більше повідомлень. Це краще реалізувати з використанням `GraphStage` (також дивіться Власна обробка за допомогою GraphStage).

```
val framing = BidiFlow.fromGraph(GraphDSL.create() { b =>
  implicit val order = ByteOrder.LITTLE_ENDIAN

  def addLengthHeader(bytes: ByteString) = {
    val len = bytes.length
    ByteString.newBuilder.putInt(len).append(bytes).result()
  }

  class FrameParser extends GraphStage[FlowShape[ByteString, ByteString]] {

    val in = Inlet[ByteString]("FrameParser.in")
    val out = Outlet[ByteString]("FrameParser.out")
    override val shape = FlowShape.of(in, out)

    override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {

      // це містить отримані, але ще не розібрані байти
      var stash = ByteString.empty
      // це містить довжину поточного повідомлення, або -1 на межі
      var needed = -1

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (isClosed(in)) run()
          else pull(in)
        }
      })
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
```

```
30.           val bytes = grab(in)
31.           stash = stash ++ bytes
32.           run()
33.         }
34.
35.         override def onUpstreamFinish(): Unit = {
36.           // або ми скінчили
37.           if (stash.isEmpty) completeStage()
38.           // або ми ще маємо байти на досилання
39.           // очікуємо завершення, та дозволяємо завершитись run()
40.           // коли залишок stash був надісланий в потік
41.           else if (isAvailable(out)) run()
42.         }
43.       })
44.
45.       private def run(): Unit = {
46.         if (needed == -1) {
47.           // чи ми на межі? тоді треба зрозуміти наступну довжину
48.           if (stash.length < 4) {
49.             if (isClosed(in)) completeStage()
50.             else pull(in)
51.           } else {
52.             needed = stash.iterator.getInt
53.             stash = stash.drop(4)
54.             run() // цикл назад щоб, до, можливо, вже надісланого наступного пакету
55.           }
56.         } else if (stash.length < needed) {
57.           // ми на середині повідомлення, треба більше байт, або ми маємо зупинитись, якщо вхід закрито
58.           if (isClosed(in)) completeStage()
59.           else pull(in)
60.         } else {
61.           // ми маємо досить, щоб надіслати щонайменьше одне повідомлення, так що робимо це
62.           val emit = stash.take(needed)
```

```
63.             stash = stash.drop(needed)
64.             needed = -1
65.             push(out, emit)
66.           }
67.         }
68.       }
69.     }
70.
71.     val outbound = b.add(Flow[ByteString].map(addLengthHeader))
72.     val inbound = b.add(Flow[ByteString].via(new FrameParser))
73.     BidiShape.fromFlows(outbound, inbound)
74. })
```

З ціма реалізаціями ми можемо побудувати стек протоколів та протестувати його:

```
1.  /* конструкція стеку протоколів
2.   *         +-------------------------------------+
3.   *         | stack                               |
4.   *         |                                     |
5.   *         |  +-------+            +---------+  |
6.   *    ~>   O~~o       |     ~>     |         o~~O    ~>
7.   * Message |  | codec | ByteString | framing |  | ByteString
8.   *    <~   O~~o       |     <~     |         o~~O    <~
9.   *         |  +-------+            +---------+  |
10.  *         +-------------------------------------+
11.  */
12. val stack = codec.atop(framing)
13.
14. // протестуємо через підключення його до своєї протилежності, та закриємо правий кінець
15. val pingpong = Flow[Message].collect { case Ping(id) => Pong(id) }
16. val flow = stack.atop(stack.reversed).join(pingpong)
17. val result = Source((0 to 9).map(Ping)).via(flow).limit(20).runWith(Sink.seq)
18. Await.result(result, 1.second) should ===((0 to 9).map(Pong))
```

Цей приклад демонструє як субграфи `BidiFlow` можуть бути поєднані разом, та також обернуті завдяки методу `.reversed`. Тест симулює обидві чистини комунікації мережевого протокола, без того, щоб відкривати мережеве з'єднання — потоки можуть з'єднуватись напряму.

## Доступ до матеріалізованого значення в Graph

В певних випадках може бути потрібним подати назад матеріалізоване значення Graph (часткового, замкненого, або що містить Source, Sink, Flow або BidiFlow). Це можливо через використання `builder.materializedValue`, що дає `Outlet`, який може бути використаний в графі як звичайне джерело або роз'єм, та що буде при нагоді видавати матеріалізоване значення. Якщо матеріалізоване значення потрібне більше ніж в одному місці, можливо викликати `materializedValue` любе число разів, щоб отримати потрібне число роз'ємів.

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

Будьте уважними, та не ввійдіть в цикл, де матеріалізоване значення насправлі додає до матеріалізованого значення. Наступний приклад демонструє випадок, де матеріалізоване `Future` з fold подається знову в сам fold.

```
1. import GraphDSL.Implicits._
2. // Це не спродукує значення:
3. val cyclicFold: Source[Int, Future[Int]] = Source.fromGraph(GraphDSL.create(Sink.fold[Int, Int](0)(_ + _))
4.   { implicit builder => fold =>
5.   // - Fold не може завершитись, докі не завершиться mapAsync
6.   // - mapAsync не може завершитись, доки не завершиться матеріалізоване Future, що виробляє fold
7.   // Як результат, цей Source ніколи нічого не видасть, та його матеріалізоване Future ніколи не завершиться
8.   builder.materializedValue.mapAsync(4)(identity) ~> fold
9.   SourceShape(builder.materializedValue.mapAsync(4)(identity).outlet)
10. })
```

## Цикли графа, життєздатність та глухі кути

Цикли в обмежених топологіях потоку потребують спеціального розгляду, щоб уникнути потенційних глухих кутів, або інших проблем життєздатності. Цей розділ показує декілька прикладів проблем, що можуть виникнути завдяки наявності зворотніх петель в обробці потокових графів.

В наступних прикладах створені робочі графи, але вони не виконуються, бо кожний має деяку проблему, та ввійде в глухий кут після запуску. Змінна `Source` не визначена за природою, та число елементів нессуттєве для описуваних проблем.

Перший приклад демонструє граф, що містить наївний цикл. Граф приймає елементи з джерела, друкує їх, потім розсилає ці елементи до споживача (ми поки що використовуємо `Sink.ignore`), та до зворотньої петлі, що зливаєтсья назад в головний потік через поєднання `Merge`.

**Зауваження**

DSL графів дозволяє зміну напрямку стрілок з'єднань, що, зокрема, корисне при створенні циклів — як ми побачимо, є випадки, коли це вкрай допомагає.

```
1. // УВАГА! Граф нижче містить глухий кут!
2. RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
3.   import GraphDSL.Implicits._
4.
5.   val merge = b.add(Merge[Int](2))
6.   val bcast = b.add(Broadcast[Int](2))
7.
8.   source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
9.             merge                    <~                      bcast
10.  ClosedShape
11. })
```

Виконавши це, ми побачимо, що після декількох надрукованих чисел, наступні елементи більше не журнулюються на консоль - вся обробка зупиняється після деякого часу. Після деякого розслідування ми знаходимо, що:

- через злиття з `source` ми збільшили число елементів в циклі
- через розсилку назад в цикл ми не зменшили число елементів в циклі

Оскільки потоки  Akka Streams (та Reactive Streams взагалі) гарантують обмежену обробку (дивіться розділ "Буферизація" для додаткових деталей), це означає, що тільки обмежене число елементів буферизується в кожний проміжок часу. Оскільки наш цикл отримує все більше і більше елементів, з часом всі його внутрішні буфери будуть заповнені, що закриє зворотнім тиском `source` назавжди. Щоб бути в змозі обробити більше елементів з `source`, треба

якось вийти з цикла.

Якщо ми модифікуємо наш цикл зворотнього зв'язку, замінивши поєднання `Merge` на `MergePreferred`, ми можемо уникнути глухого кута. `MergePreferred` нерівне, оскільки завжди намагається споживати з преоритетного вхідного порта, якщо там є елементи, перед спробою інших вхідних портів з низьким преоритетом. Оскільки ми замикаємо цикл через преоритетний порт, завжди є гарантія, що елементи в циклі будуть рухатись.

```
1. // УВАГА! Граф нижче зупиняється, споживаючи з "source" після декількох кроків
2. RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
3.   import GraphDSL.Implicits._
4.
5.   val merge = b.add(MergePreferred[Int](1))
6.   val bcast = b.add(Broadcast[Int](2))
7.
8.   source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
9.             merge.preferred              <~                  bcast
10.  ClosedShape
11. })
```

Якщо ми виконаємо цей приклад, ми побачимо ту ж послідовність чисел, надруковану знову і знову, але обробка не зупиняється. Таким чином, ми уникаємо глухого кута, але `source` все ще під зворотнім тиском назавжди, оскільки простір буфера ніколи не відновлюється: єдина дія, що ми бачимо, є циркуляція пари початкових елементів з `source`.

**Зауваження**

Ми тут бачимо, що в певних випадках нам треба обрати між обмеженістю та життєспроможністю. Наш перший приклад міг би не попасти в глухий кут, якби міг би бути безмежний буфер в циклі. Або навпаки, якби елементи в циклі були б збалансовані (скільки елементів видаляється, стільки і додаються), тоді не було б глухого кута.

Щоб зробити наш граф одночасно живим (без глухого кута), та чесним, ми можемо ввести відкидання елементів на зворотньому шляху. В цьому випадку ми обираємо операцію `buffer()`, надаючи стратегію відкидання `OverflowStrategy.dropHead`.

```
1. RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
2.   import GraphDSL.Implicits._
3.
```

```
  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
})
```

Якщо ми виконаємо цей приклад, ми побачимо що

- Потік елементів не зупиняється, завжди друкуються елементи
- Ми бачимо, що деякі числа друкуються декілька разів (через зворотній цикл), але в середньому в довшій перспективі числа зростають

Цей приклад висвічує те, що одне рішення для уникання грухих кутів за наявності потенційно незбалансованих циклів (циклів, де число циркулюючих елементів необмежене) є відкидання елементів. Альтернативою може бути визначення більших буферів з `OverflowStrategy.fail`, що буде давати збій потоку замість глухого блокування, якщо врешті решт простір буде спожито.

Як ви визначили в попередніх прикладах, коренева проблема була в незбалансованій природі зворотнього цикла. Ми обійшли цю проблему, додавши відкидання елементів, але тепер ми бажаємо побудувати цикл, що збалансований з самого початку. Щоб досягти цього, ми можифікували наш перший граф, замінивши поєднання `Merge` на `ZipWith`. Оскільки `ZipWith` приймає один елемент з `source` *та* з зворотньої петлі, щоб ввести один елемент в цикл, ми керуємо балансуванням елементів.

```
// УВАГА! Граф нижче ніколи не обробляє жодного елемента
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val zip = b.add(ZipWith[Int, Int, Int]((left, right) => right))
  val bcast = b.add(Broadcast[Int](2))

  source ~> zip.in0
  zip.out.map { s => println(s); s } ~> bcast ~> Sink.ignore
  zip.in1             <~                bcast
  ClosedShape
})
```

І знову, коли ми намагаємось виконати приклад, з'ясрвується, що він взагалі не друкує жодного елемента! Після деякого дослідження ми розуміємо, що:

- Щоб отримати перший елемент з `source` в циклі нам треба вже існуючий елемент в циклі
- Щоб отримати перший елемент в циклі, нам треба елемент з `source`

Ці дві умови є типовою проблемою "куриця-або-яйце". Рішення є ввести первинний елементв цикл, що є незалежним від `source`. Ми робимо це, використовуючи поєднання `Concat` на зворотній петлі, що вводить єдиний елемент з використанням `Source.single`.

```
1.  RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
2.    import GraphDSL.Implicits._
3.
4.    val zip = b.add(ZipWith((left: Int, right: Int) => left))
5.    val bcast = b.add(Broadcast[Int](2))
6.    val concat = b.add(Concat[Int]())
7.    val start = Source.single(0)
8.
9.    source ~> zip.in0
10.   zip.out.map { s => println(s); s } ~> bcast ~> Sink.ignore
11.   zip.in1 <~ concat <~ start
12.               concat         <~          bcast
13.   ClosedShape
14. })
```

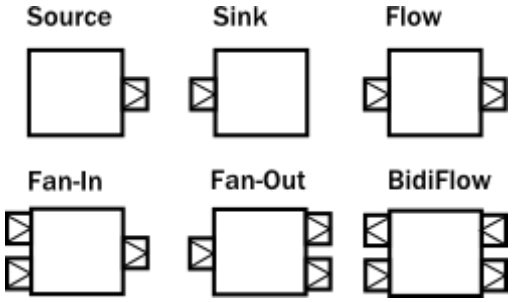
Коли ми виконуємо цей приклад, ми бачимо, що обробка починається, та ніколи не закінчується. Важлий висновок з цього приклада в тому, що збалансовані цикли часто потребують вставки в цикл початкового "розкруточного" елемента.

# Модульність, композиція та ієрархія

Потоки Akka Streams провадять одноманітну модель для обробки графів, що дозволяє гнучку композицію повторно використовуваних компонент. В цій главі ми покажемо, як це виглядає з концептуальної перспективи, та з боку зору API, демонструючи модулярні аспекти бібліотеки.

## Основи композиції та модульності

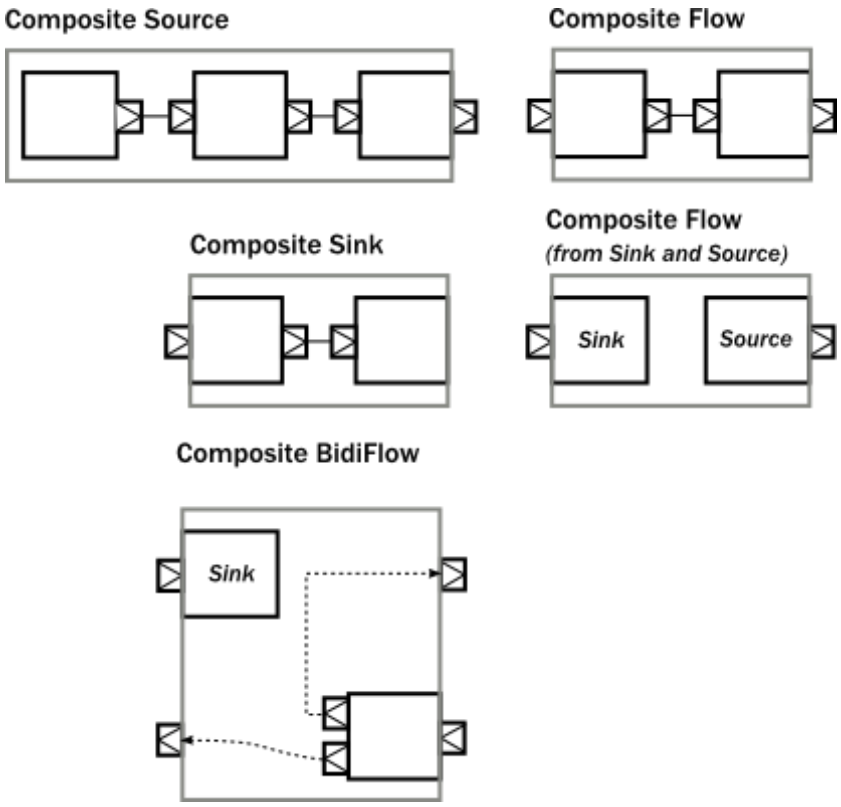
Кожна стадія обробки, що використовується в  Akka Streams, може бути уявлена як "ящик" з входом та виходом, де елементи, що мають бути оброблені, надходять та звідки виходять. В цьому погляді `Source` є нічим іншим, ніж "ящиком" з одним вихідним портом, або `BidiFlow` є "ящик" з точно двома вхідними та вихідними портами. На малюнку нижче ми ілюструємо найбільш загально використовувані стадії в вигляді "ящиків".




*Лінійні* стадії є `Source`, `Sink` та `Flow`, як вони можуть бути використані для композиції прямих ланцюжків стадій обробки. Стадії поєднання та розгалуження мають декілька входних або багато виходних портів, та, таким чином, вони дозволяють будувати більш складні фрми графів, не тільки ланцюжки. Стадії `BidiFlow` звичайно корисні в задачах IO, де мають бути оброблені вхідні та вихідні канали. Через специфічну форму `BidiFlow` просто накласти їх один на одний, щоб побудувати, наприклад, розшарений протокол. Підтримка `TLS` в Akka реалізована через `BidiFlow`.

Ці повторно використовувані компоненти дозволяють створювати складні мережі обробки. Однак те, що ми бачили дотепер, не реалізує модулярності. Бажано, наприклад, спакувати більший граф-сутність в повторно використовуваний компонент, що приховує свої нутрощі, показуючи тільки порти, що призначені для використання користувачами модуля. Одним простим прикладом є компонент `Http` сервера, що зсерединини закодований як `BidiFlow`, що має інтерфейс з клієнтським TCP з'єднанням з використанням пари портів вводу-виводу, що приймають та надсилають `ByteString`, тоді як вищі порти надсилають та видають примірники `HttpRequest` та `HttpResponse`.

Наступний малюнок демонструє разні композитні стадії, що містять різні інші типи стадй в собі, але приховують їх за *фігурою*, що виглядає як `Source`, `Flow`тощо, .

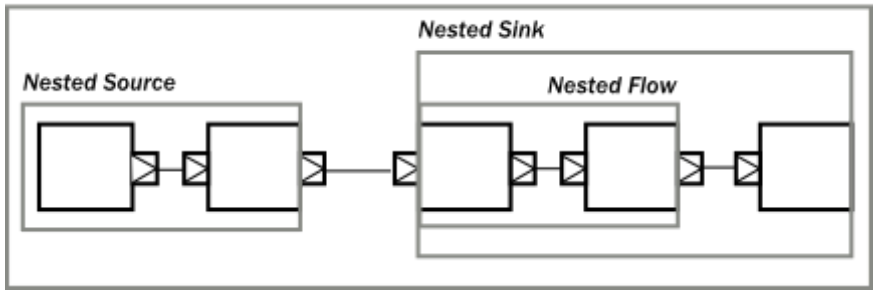



Одним цікавим прикладом вище є `Flow`, який складається з роз'єднаних `Sink` та `Source`. Це можна досягнути через використання метода конструктора `fromSinkAndSource()` на `Flow`, що приймає дві частини як параметри.

Приклад `BidiFlow` демонструє, що зсередини модуль може мати довільну складність, та показані порти можуть бути під'єднані в гнучкий спосіб. Єдине обмеження в тому, що всі порти оточуючих модулів мають бути або під'єднані один до одного, або показані як інтерфейсні порти, та число таких портів має співпадати потребам фігури, наприклад, `Source` дозволяє тільки один відкритий порт, залишок внутрішніх портів мусять бути відповідно під'єднані.

Ця механіка дозволяє довільно вкладати модулі. Наприклад, наступна фігура демонструє `RunnableGraph`, що побудований з композитного `Source` та композитного `Sink` (що, в свою чергу, містить композитний `Flow`).

RunnableGraph




Диаграма вище містить ще одну фігуру, що ми не бачили дотепер, що називається `RunnableGraph`. Як з'ясовується, якщо ми з'єднаємо всі показані порти разом, так що не залишиться жодного відкритого порта, тоді ми отримаємо *замкнений* модуль. Це саме те, що представляє клас `RunnableGraph`. Це фігура, що може прийняти `Materializer`, та перетворити на мережу виконуваних сутностей, що виконують описане завдання. Фактично, `RunnableGraph` є сам по собі модулем, та (може трохи несподівано) може використовуватись як частина більших графів. Це рідко використовується, коли замкнена фігруа-граф трапляється в більшому графі (оскільки вона стає ізольованим островом, бо не має відкритих портів для комунікації з рештою графа), але це демонструє уніформнту підлежну модель.

Якщо ми спробуємо побудувати фрагмент кода, що відповідає діаграмі вище, наша перша спроба може виглядати так:

```
1. Source.single(0)
2.   .map(_ + 1)
3.   .filter(_ != 0)
4.   .map(_ - 2)
5.   .to(Sink.fold(0)(_ + _))
6.
7. // ... де тут вкладання?
```

Однак ясно, що в нашй першій спрбі  немає присутнього вкладання, оскільки бібліотека не може визначити, де прокласти межу для составного модуля, бо це є наша відповідальність. Якщо ми використовуємо DSL, запроваджений через класи `Flow`, `Source`, `Sink`, тоді вкладенність може бути досягнена через виклик одного з методів `withAttributes()` або `named()` (де останній є лише скороченням до додавання атрибута метода).

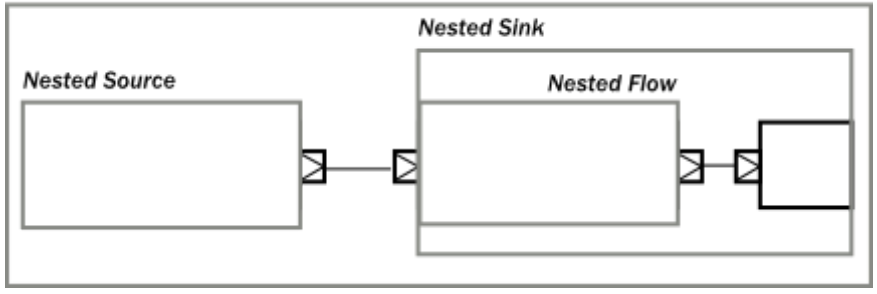
Наступний код демонструє, як досягти це баження вкладання:

```
1. val nestedSource =
2.   Source.single(0) // Атомарне джерело
```

```
    .map(_ + 1) // атомарна стадія обробки
    .named("nestedSource") // огортає поточне Source, та дає йому ім'я

val nestedFlow =
  Flow[Int].filter(_ != 0) // атомарна стадія обробки
    .map(_ - 2) // інша атомарна стадія обробки
    .named("nestedFlow") // огортає Flow, та дає йому ім'я

val nestedSink =
  nestedFlow.to(Sink.fold(0)(_ + _)) // з'єднуємо атомарний приймач до nestedFlow
    .named("nestedSink") // огортаємо його

// створюємо RunnableGraph
val runnableGraph = nestedSource.to(nestedSink)
```

Як тільки ми приховали нутрощі наших компонент, вони діють як інші вбудовані компоненти подібної форми. Якщо ми приховуємо деякі нутрощі наших композитів, результат виглядає так само, якби був використаний любий інший попередньо визначений компонент:

**RunnableGraph**




Якщо ми подивимось на використання вбудованих компонент, та наших власних компонент, не буде різниці в використанні, як продемонстровано в коді нижче.

```
// Створити RunnableGraph з компонент
val runnableGraph = nestedSource.to(nestedSink)

```

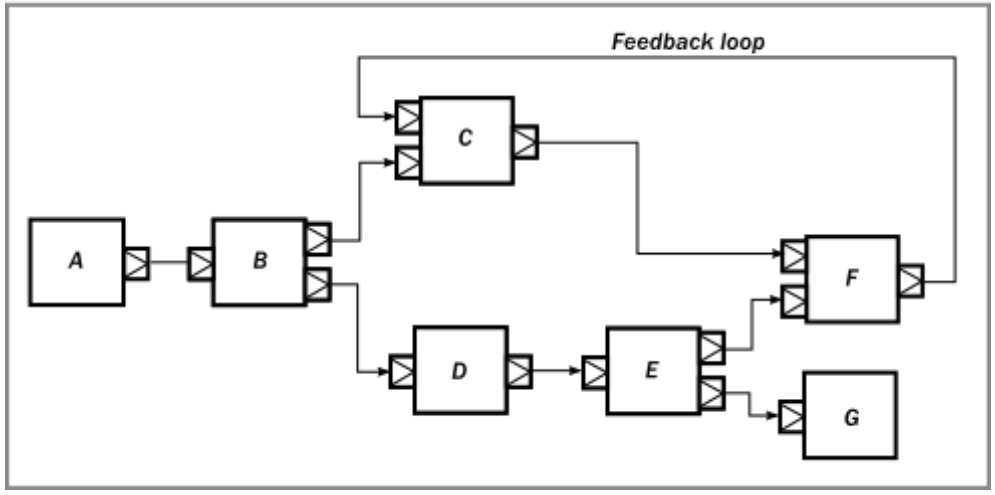
```
4. // Використання уніформне, не має значенн, чи модулі є атомарними або композитними
5. val runnableGraph2 = Source.single(0).to(Sink.fold(0)(_ + _))
```

## Композиція складних систем

В попередньому розділі ми дослідили можливість композиції та створення ієрархії, але ми стояли осторонь нелінійних, узагальнених компонент графів. Однак в Akka Streams немає нічого, що б змушувало обробку потоків бути тільки лінійною. DSL для `Source` та його друзів оптимізоване для створення таких лінійних ланцюжків, бо вони є самими загальними на практиці. Існує більш складний DSL для будування складних графів, що може бути використане, якщо потрібна більша гнучкість. Ми побачимо, що різниця між двома DSL існує тільки на поверхні: концепції, якими вони оперують, є уніформними між всіма DSL та гарно підходять один одному.

Як перший приклад давайте подивимось на більш складне розташування:

**RunnableGraph**




Діаграма показує `RunnableGraph` (як ви пам'ятаєте, якщо немає непідключених портів, граф є замкнений, і, таким чином, може бути матеріалізоване), що інкапсулює нетривіальну мережу обробки потока. Він містить стадії поєднання, розгалуження, прямі та непрямі цикли. Метод `runnable()` об'єкта `GraphDSL` дозволяє створення загального, замкненого, та виконуваного графа. Наприклад, мережа на діаграмі може бути реалізована таким чином:

```
1. import GraphDSL.Implicits._
```

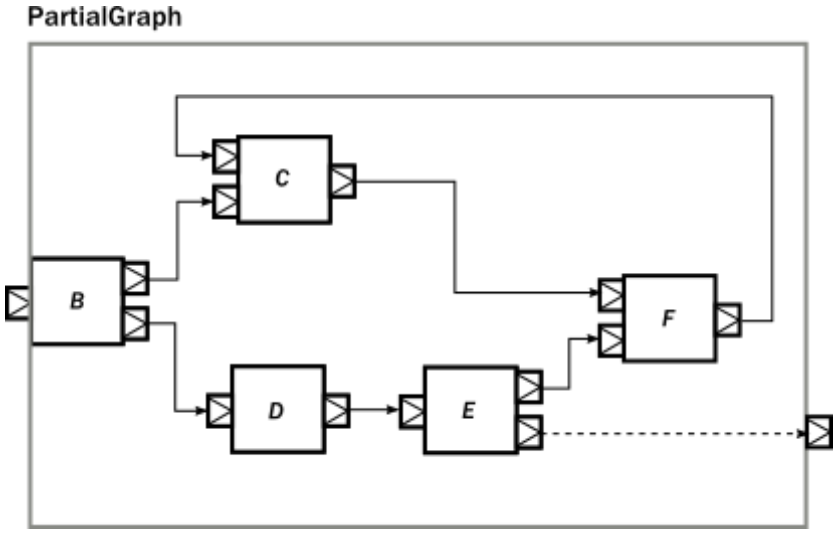
```
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val A: Outlet[Int]                  = builder.add(Source.single(0)).out
  val B: UniformFanOutShape[Int, Int] = builder.add(Broadcast[Int](2))
  val C: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val D: FlowShape[Int, 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
})
```

В коді вище ми використали неявну нумерацію портів (щоб зробити граф більш читаємим та подібним до діаграми), та ми імпортували `Source`, `Sink` та `Flow` явно. Можливо посилатись на порти явно, на немає необхідності імпортувати наші лінійні стадії через `add()`, так що інша версія може виглядати так:

```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(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))

  Source.single(0) ~> B.in; B.out(0) ~> C.in(1); C.out ~> F.in(0)
  C.in(0) <~ F.out

  B.out(1).map(_ + 1) ~> E.in; E.out(0) ~> F.in(1)
  E.out(1) ~> Sink.foreach(println)
  ClosedShape
```

```
14. })
```

Подібно до випадку в першому розділі, до тепер ми не розглядали модулярність. Ми створили складний граф, але розкладка пласка, не модуляризована. Ми будемо модифікувати наш приклад та створимо повторно використовуваний компонент за допомогою DSL графів. Шлях зробити це - використати метод-фабрику `create()` на `GraphDSL`. Якщо ми видалимо джерела та приймачі з попереднього приклада, те що залишиться буде частковим графом:



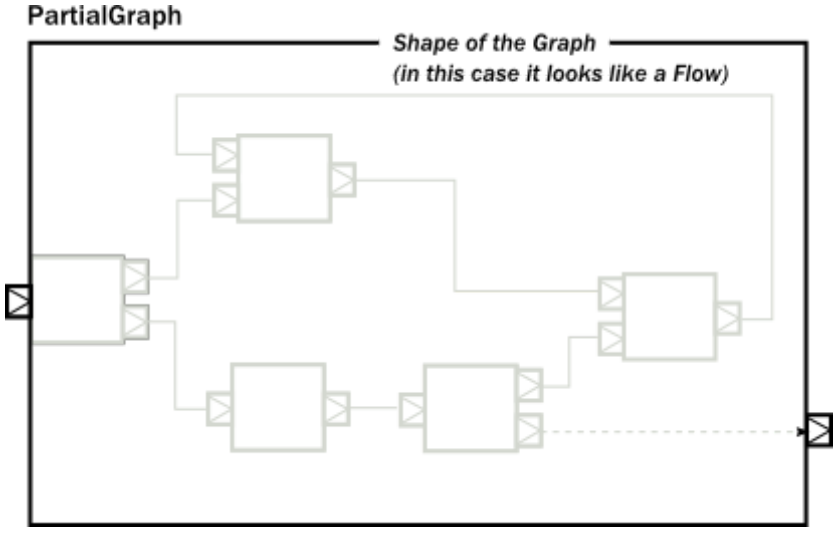

Ви можете створити граф в коді з використанням DSL в спосіб, подібний до попереднього:

```
1.  import GraphDSL.Implicits._
2.  val partial = GraphDSL.create() { implicit builder =>
3.    val B = builder.add(Broadcast[Int](2))
4.    val C = builder.add(Merge[Int](2))
5.    val E = builder.add(Balance[Int](2))
6.    val F = builder.add(Merge[Int](2))
7.
8.                                    C  <~  F
9.    B  ~>                           C  ~>  F
10.   B  ~>  Flow[Int].map(_ + 1)  ~>  E  ~>  F
```

```
11.   FlowShape(B.in, E.out(1))
12. }.named("partial")
```

Єдиний новий додаток є повернуте значення блока побудовника, що є `Shape`. Всі графи (включаючи `Source`, `BidiFlow` тощо) мають форму, що кодує *типовані* порти модуля. В нашому прикладі залишився тільки один вхідний та вихідний порти, так що ми можемо декларувати його як `FlowShape`, через повернення його примірника. Хоча це досить можливо створити нові типи `Shape`, зазвичай рекомендовано використовувати одну з підходячих вбудованих форм.

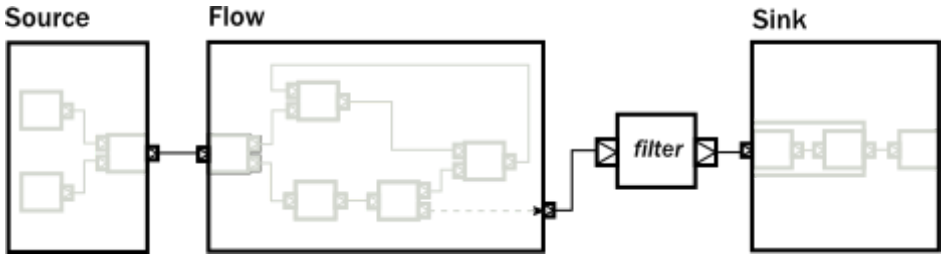
Отриманий граф є вже відповідно огорнутим модулем, так що немає потреби викликати named() для інкапсуляції графа, але є гарною практикою надавати імена модулям, що допомагатиме в налаштуванні.




Оскільки наш частковий граф має правильну форму, він може Since our partial graph has the right shape, it can be already used in the simpler, linear DSL:

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

Хоча його все ще не можна використовувати як `Flow`, (тобто, доки ми не можемо викликати `.filter()` на ньому), але `Flow` має метод `fromGraph()`, що тільки додає DSL до `FlowShape`. Існують подібні методи для `Source`, `Sink` та `BidiShape`, так що є простим повернутись до простішого DSL, якщо граф має правильну форму. Для зручності також можливо проскочити створення часткового графа, та використати один зі зручних методів створення. Щоб продемонструвати це ми створими наступний граф:




Версія кода наведеного замкненого графа може виглядати так:

```
1. // Конвертувати частковий граф FlowShape на Flow для отримання доступу
2. // до рідкого DSL (наприклад, щоб бути в змозі викликати .filter())
3. val flow = Flow.fromGraph(partial)
4.
5. // Простий спосіб створити граф для  Source
6. val source = Source.fromGraph( GraphDSL.create() { implicit builder =>
7.   val merge = builder.add(Merge[Int](2))
8.   Source.single(0)      ~> merge
9.   Source(List(2, 3, 4)) ~> merge
10.
11.   // Показуєм тільки один вихідний порт
12.   SourceShape(merge.out)
13. })
14.
15. // Побудова Sink з вкладеним Flow, з використанням рідкого DSL
16. val sink = {
17.   val nestedFlow = Flow[Int].map(_ * 2).drop(10).named("nestedFlow")
18.   nestedFlow.to(Sink.head)
19. }
20.
21. // Складаємо все докупи
22. val closed = source.via(flow.filter(_ > 1)).to(sink)
```

**Зауваження**

Всі розділи побудови графів перевіряють, чи отриманий граф має всі порти підключеними, за винятком показаних, та буде викликати виключення, якщо це порушується.

Ми все ще можемо продемонструвати, що `RunnableGraph` є компонентом, таким самим, як любий інший, що може бути бути вставлений в графи. В наступному фрагменті ми вбудовуємо один замкнений граф в інший:

```
1. val closed1 = Source.single(0).to(Sink.foreach(println))
2. val closed2 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
3.   val embeddedClosed: ClosedShape = builder.add(closed1)
4.   // …
5.   embeddedClosed
6. })
```

Тип імпортованого модуля вказує, що імпортований модуль має `ClosedShape`, так що ми не в змозі під'єднати будь що інше в замкнений граф. Тим не менш, цей "острів" вбудований відповідним чином, та буде матеріалізований як любий інший модуль, що є частиною графа.

Як ми продемонстрували, два DSL є повністю взаємооперабельні, бо кодують подібні вкладені структури "ящиків з портами", та де DSL відрізняються, це в тому, що кожний намагається бути найбільш потужним на різних рівнях абстракції. Можливо вбудувати складні графи в рідкий DSL, та це так само просто імпортувати та вбудувати `Flow` тощо, в більшу, складну структуру.

Також, як ми бачили, що кожний модуль має `Shape` (наприклад, `Sink` має `SinkShape`), незалежно, який DSL був використаний для його створення. Це одноманітне предсталення дозволяє багату компоновку різних сутностей обробки потоку в зручний спосіб.

## Матеріалізовані значення

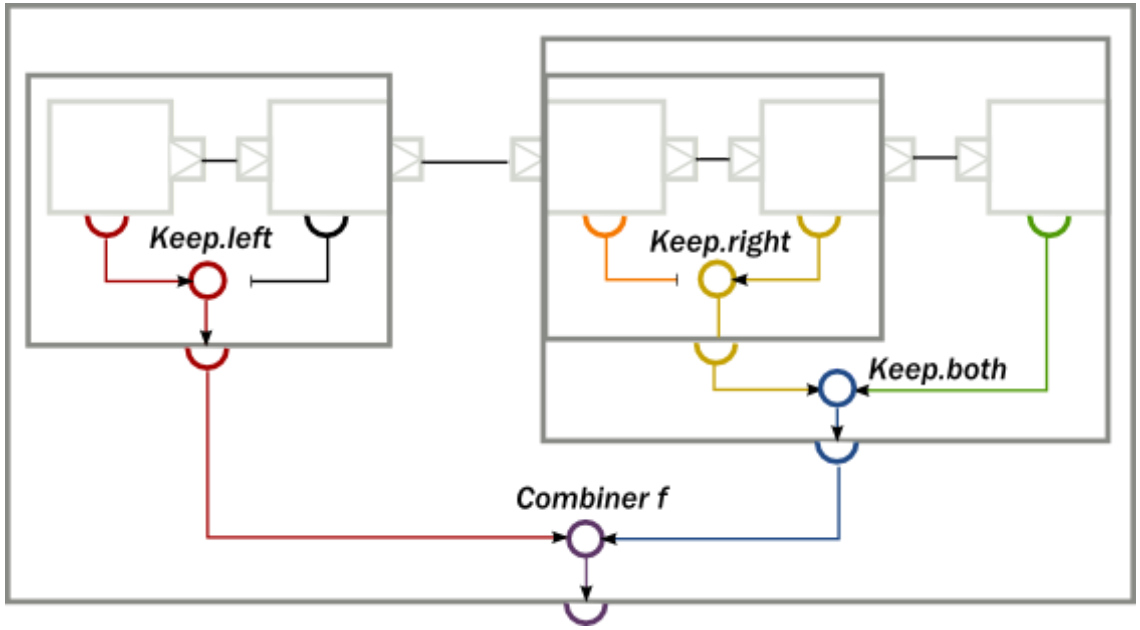
Після усвідомлення, що `RunnableGraph` нічого більшого, ніж модуль без невикористаних портів (він є островом), стає зрозумілим, що після матеріалізації єдиним шляхом взаємодіяти з роблячою логікою потока, що виконується, є деякий побічний канал. Цей побічний канал представлений як *матеріалізоване значення*. Ситуація подібна до акторів `Actor`, де примірник `Props` описує логіку актора, але це виклик до `actorOf()`, що створює насправді роблячого актора, та повертає `ActorRef`, що може бути використаний для комунікації з самим роблячим актором. Оскільки `Props` може бути повторо використане, кожний виклик буде повертати інше посилання.

Коли йдеться про потоки, кожна матеріалізація створює нову виконувану мережу, що відповідає схемі, закодованій в наданому `RunnableGraph`. Щоб бути в змозі взаємодіяти з виконуваною мережею, кожна матеріалізація має повертати різний об'єкт, що провадить потрібні можливості взаємодії. Іншими словами, `RunnableGraph` можна розглядати як фабрику, що створює:

- мережу роблячих сутностей обробки, до яких немає доступа ззовні
- матеріалізоване значення, що опціонально провадить контрольовану можливість взаємодії з мережею

Однак на відміну від акторів, можна зі стадій обробки може виробляти матеріалізоване значення, так що, коли ми компонуємо декілька стадій або модулей, нам треба скомбінувати матеріалізоване значення також (є правила по замовчанню, що роблять це простішим, наприклад, to() та via() пілкується про найбільш загальний випадок, беручи матеріалізоване значення зліва. Дивіться Комбінація матеріалізованих значень щодо деталей). Ми демонструємо, як це робить, через приклад кода, та діаграму, що графічно демонструє що відбувається.

Просування індивідуальних матеріалізованих значень з замкнених модулів далі нагору буде виглядати таким чином:




Щоб реалізувати показане вище, спочатку ми створюємо композитний `Source`, да замкнений `Source` має матеріалізований тип `Promise[Unit]`. Через використання функції комбінатора `Keep.left`, результуючий матеріалізований тип є типом вкладеного модуля (показаний червоним на діаграмі):

```
1. // Матеріалізується до Promise[Option[Int]]   (червоне)
```

```
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]

// Матеріалізується до Unit    (чорне)
val flow1: Flow[Int, Int, NotUsed] = Flow[Int].take(100)

// Матеріалізується до Promise[Int] (червоне)
val nestedSource: Source[Int, Promise[Option[Int]]] =
  source.viaMat(flow1)(Keep.left).named("nestedSource")
```

Далі ми створюємо композитний `Flow` з двох менших компонент. Тут другий замкнений `Flow` має матеріалізований тип `Future[OutgoingConnection]`, та ми просуваємо його до батька через використання `Keep.right` в якості функції комбінатора (показане *жовтим* на діаграмі):

```
// Матеріалізоване до Unit (помаранчевий)
val flow2: Flow[Int, ByteString, NotUsed] = Flow[Int].map { i => ByteString(i.toString) }

// Матеріалізується до Future[OutgoingConnection] (жовтий)
val flow3: Flow[ByteString, ByteString, Future[OutgoingConnection]] =
  Tcp().outgoingConnection("localhost", 8080)

// Матеріалізується до Future[OutgoingConnection] (жовтий)
val nestedFlow: Flow[Int, ByteString, Future[OutgoingConnection]] =
  flow2.viaMat(flow3)(Keep.right).named("nestedFlow")
```

Як третій крок, ми створюємо композитний `Sink` використовуючи наш `nestedFlow` як будівельний блок. В цьому фрагменті, обоє, замкнений `Flow` та наступний `Sink` мають матеріалізоване значення, що цікаві для нас, так що ми використовуємо `Keep.both` щоб отримати `Pair` з них, як матеріалізований тип `nestedSink` (вказаний *синім* кольором на діаграмі)

```
// Матеріалізується до Future[String] (зелений)
val sink: Sink[ByteString, Future[String]] = Sink.fold("")(_ + _.utf8String)

// Матеріалізується до (Future[OutgoingConnection], Future[String]) (синій)
val nestedSink: Sink[Int, (Future[OutgoingConnection], Future[String])] =
```

```
  nestedFlow.toMat(sink)(Keep.both)
```

Як останній приклад, ми з'єднуємо разом `nestedSource` та `nestedSink`, та ми використовуємо власну функцію-комбінатор для створення ще одного матеріалізованого типа отриманого `RunnableGraph`. Ця функція-комбінаторпросто ігнорує частину `Future[Sink]`, та огортає інші два значення у власний кейс клас `MyClass` (вказаний *фіолетовим* кольором на діаграмі):

```
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, _))
}

// Матеріазується до Future[MyClass] (фіолетовий)
val runnableGraph: RunnableGraph[Future[MyClass]] =
  nestedSource.toMat(nestedSink)(f)
```

**Зауваження**

Вкладені структури в прикладі вище не потрібні для комбінування матеріалізованих значень, вони тільки демонструють, як дві можливості роблять разом. Дивіться Комбінування матаріалізованих значень для подальших прикладів комбінування матеріалізованих значень без задіяних вкладань та ієрархії.

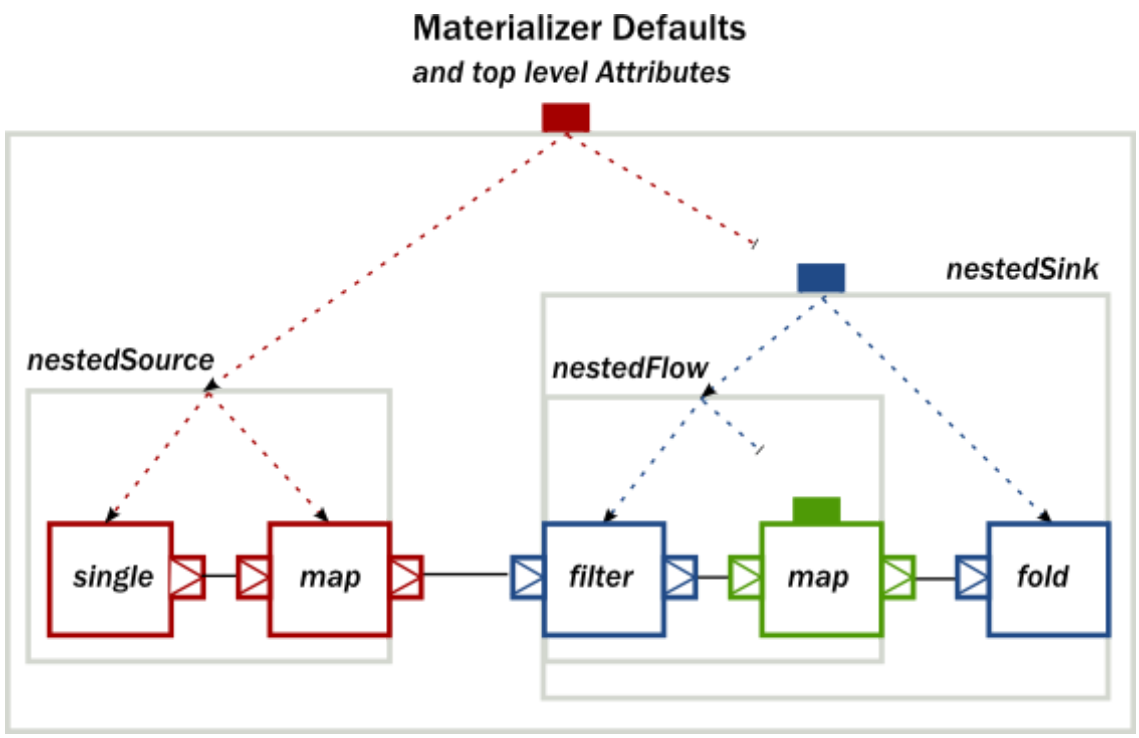
## Attributes

We have seen that we can use `named()` to introduce a nesting level in the fluid DSL (and also explicit nesting by using `create()` from `GraphDSL`). Apart from having the effect of adding a nesting level, `named()` is actually a shorthand for calling `withAttributes(Attributes.name("someName"))`. Attributes provide a way to fine-tune certain aspects of the materialized running entity. For example buffer sizes for asynchronous stages can be controlled via attributes (seeBuffers for asynchronous stages). When it comes to hierarchic composition, attributes are inherited by nested modules, unless they override them with a custom value.

The code below, a modification of an earlier example sets the `inputBuffer` attribute on certain modules, but not on others:

```
1. import Attributes._
2. val nestedSource =
3.   Source.single(0)
4.     .map(_ + 1)
5.     .named("nestedSource") // Wrap, no inputBuffer set
6.
7. val nestedFlow =
8.   Flow[Int].filter(_ != 0)
9.     .via(Flow[Int].map(_ - 2).withAttributes(inputBuffer(4, 4))) // override
10.    .named("nestedFlow") // Wrap, no inputBuffer set
11.
12. val nestedSink =
13.   nestedFlow.to(Sink.fold(0)(_ + _)) // wire an atomic sink to the nestedFlow
14.    .withAttributes(name("nestedSink") and inputBuffer(3, 3)) // override
```

The effect is, that each module inherits the `inputBuffer` attribute from its enclosing parent, unless it has the same attribute explicitly set. `nestedSource` gets the default attributes from the materializer itself. `nestedSink` on the other hand has this attribute set, so it will be used by all nested modules. `nestedFlow` will inherit from `nestedSink` except the `map` stage which has again an explicitly provided attribute overriding the inherited one.




This diagram illustrates the inheritance process for the example code (representing the materializer default attributes as the color *red*, the attributes set on `nestedSink` as *blue* and the attributes set on `nestedFlow` as *green*).

# Buffers and working with rate

When upstream and downstream rates differ, especially when the throughput has spikes, it can be useful to introduce buffers in a stream. In this chapter we cover how buffers are used in Akka Streams.

## Buffers for asynchronous stages

In this section we will discuss internal buffers that are introduced as an optimization when using asynchronous stages.

To run a stage asynchronously it has to be marked explicitly as such using the `.async` method. Being run asynchronously means that a stage, after handing out an element to its downstream consumer is able to immediately process the next message. To demonstrate what we mean by this, let's take a look at the following example:

```
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)
```

Running the above example, one of the possible outputs looks like this:

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

Note that the order is *not* `A:1, B:1, C:1, A:2, B:2, C:2,` which would correspond to the normal fused synchronous execution model of flows where an element completely passes through the processing pipeline before the next element enters the flow. The next element is processed by an asynchronous stage as soon as it is emitted the previous one.

While pipelining in general increases throughput, in practice there is a cost of passing an element through the asynchronous (and therefore thread crossing) boundary which is significant. To amortize this cost Akka Streams uses a*windowed*, *batching* backpressure strategy internally. It is windowed because as opposed to a Stop-And-Wait protocol multiple elements might be "in-flight" concurrently with requests for elements. It is also batching because a new element is not immediately requested once an element has been drained from the window-buffer but multiple elements are requested after multiple elements have been drained. This batching strategy reduces the communication cost of propagating the backpressure signal through the asynchronous boundary.

While this internal protocol is mostly invisible to the user (apart form its throughput increasing effects) there are situations when these details get exposed. In all of our previous examples we always assumed that the rate of the processing chain is strictly coordinated through the backpressure signal causing all stages to process no faster than the throughput of the connected chain. There are tools in Akka Streams however that enable the rates of different segments of a processing chain to be "detached" or to define the maximum throughput of the stream through external timing sources. These situations are exactly those where the internal batching buffering strategy suddenly becomes non-transparent.

## Internal buffers and their effect

As we have explained, for performance reasons Akka Streams introduces a buffer for every asynchronous processing stage. The purpose of these buffers is solely optimization, in fact the size of 1 would be the most natural choice if there would be no need for throughput improvements. Therefore it is recommended to keep these buffer sizes small, and increase them only to a level suitable for the throughput requirements of the application. Default buffer sizes can be set through configuration:

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

Alternatively they can be set by passing a `ActorMaterializerSettings` to the materializer:

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

If the buffer size needs to be set for segments of a `Flow` only, it is possible by defining a separate `Flow` with these attributes:

```
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
```

Here is an example of a code that demonstrate some of the issues caused by internal buffers:

```
import scala.concurrent.duration._
case class Tick()

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

  // this is the asynchronous stage in this graph
  val zipper = b.add(ZipWith[Tick, Int, Int]((tick, count) => count).async)
```

```
9.
10.   Source.tick(initialDelay = 3.second, interval = 3.second, Tick()) ~> zipper.in0
11.
12.   Source.tick(initialDelay = 1.second, interval = 1.second, "message!")
13.     .conflateWithSeed(seed = (_) => 1)((count, _) => count + 1) ~> zipper.in1
14.
15.   zipper.out ~> Sink.foreach(println)
16.   ClosedShape
17. })
```

Running the above example one would expect the number *3* to be printed in every 3 seconds (the `conflateWithSeed` step here is configured so that it counts the number of elements received before the downstream `ZipWith` consumes them). What is being printed is different though, we will see the number *1*. The reason for this is the internal buffer which is by default 16 elements large, and prefetches elements before the `ZipWith` starts consuming them. It is possible to fix this issue by changing the buffer size of `ZipWith` (or the whole graph) to 1. We will still see a leading 1 though which is caused by an initial prefetch of the `ZipWith` element.

**Note**

In general, when time or rate driven processing stages exhibit strange behavior, one of the first solutions to try should be to decrease the input buffer of the affected elements to 1.

## Buffers in Akka Streams

In this section we will discuss *explicit* user defined buffers that are part of the domain logic of the stream processing pipeline of an application.

The example below will ensure that 1000 jobs (but not more) are dequeued from an external (imaginary) system and stored locally in memory - relieving the external system:

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

The next example will also queue up 1000 jobs locally, but if there are more jobs waiting in the imaginary external systems, it makes space for the new element by dropping one element from the *tail* of the buffer. Dropping from the tail is a very common strategy but it must be noted that this will drop the *youngest* waiting job. If some "fairness" is desired in the sense that we want to be nice to jobs that has been waiting for long, then this option can be useful.

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

Instead of dropping the youngest element from the tail of the buffer a new element can be dropped without enqueueing it to the buffer at all.

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

Here is another example with a queue of 1000 jobs, but it makes space for the new element by dropping one element from the *head* of the buffer. This is the *oldest* waiting job. This is the preferred strategy if jobs are expected to be resent if not processed in a certain period. The oldest element will be retransmitted soon, (in fact a retransmitted duplicate might be already in the queue!) so it makes sense to drop it first.

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

Compared to the dropping strategies above, dropBuffer drops all the 1000 jobs it has enqueued once the buffer gets full. This aggressive strategy is useful when dropping jobs is preferred to delaying jobs.

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

If our imaginary external job provider is a client using our API, we might want to enforce that the client cannot have more than 1000 queued jobs otherwise we consider it flooding and terminate the connection. This is easily achievable by the error strategy which simply fails the stream once the buffer gets full.

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

# Rate transformation

## Understanding conflate

When a fast producer can not be informed to slow down by backpressure or some other signal, `conflate` might be useful to combine elements from a producer until a demand signal comes from a consumer.

Below is an example snippet that summarizes fast stream of elements to a standart deviation, mean and count of elements that have arrived while the stats have been calculated.

```
1. val statsFlow = Flow[Double]
2.   .conflateWithSeed(Seq(_))(_ :+ _)
3.   .map { s =>
4.     val μ = s.sum / s.size
5.     val se = s.map(x => pow(x - μ, 2))
6.     val σ = sqrt(se.sum / se.size)
7.     (σ, μ, s.size)
8.   }
```

This example demonstrates that such flow's rate is decoupled. The element rate at the start of the flow can be much higher that the element rate at the end of the flow.

Another possible use of `conflate` is to not consider all elements for summary when producer starts getting too fast. Example below demonstrates how `conflate` can be used to implement random drop of elements when consumer is not able to keep up with the producer.

```
1. val p = 0.01
2. val sampleFlow = Flow[Double]
3.   .conflateWithSeed(Seq(_)) {
4.     case (acc, elem) if Random.nextDouble < p => acc :+ elem
5.     case (acc, _)                             => acc
6.   }
7.   .mapConcat(identity)
```

## Understanding expand

Expand helps to deal with slow producers which are unable to keep up with the demand coming from consumers. Expand allows to extrapolate a value to be sent as an element to a consumer.

As a simple use of `expand` here is a flow that sends the same element to consumer when producer does not send any new elements.

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

Expand also allows to keep some state between demand requests from the downstream. Leveraging this, here is a flow that tracks and reports a drift between fast consumer and slow producer.

```
1. val driftFlow = Flow[Double]
2.   .expand(i => Iterator.from(0).map(i -> _))
```

Note that all of the elements coming from upstream will go through `expand` at least once. This means that the output of this flow is going to report a drift of zero if producer is fast enough, or a larger drift otherwise.

# Dynamic stream handling

## Controlling graph completion with KillSwitch

A `KillSwitch` allows the completion of graphs of `FlowShape` from the outside. It consists of a flow element that can be linked to a graph of `FlowShape` needing completion control. The `KillSwitch` trait allows to complete or fail the graph(s).

```
1. trait KillSwitch {
2.   /**
3.    * After calling [[KillSwitch#shutdown()]] the linked [[Graph]]s of [[FlowShape]] are completed normally.
4.    */
5.   def shutdown(): Unit
6.   /**
7.    * After calling [[KillSwitch#abort()]] the linked [[Graph]]s of [[FlowShape]] are failed.
8.    */
9.   def abort(ex: Throwable): Unit
10. }
```

After the first call to either `shutdown` or `abort`, all subsequent calls to any of these methods will be ignored. Graph completion is performed by both

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

A `KillSwitch` can control the completion of one or multiple streams, and therefore comes in two different flavours.

## UniqueKillSwitch

`UniqueKillSwitch` allows to control the completion of **one** materialized `Graph` of `FlowShape`. Refer to the below for usage examples.

- **Shutdown**

```
1. val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.backpressure)
2. val lastSnk = Sink.last[Int]
3.
4. val (killSwitch, last) = countingSrc
5.   .viaMat(KillSwitches.single)(Keep.right)
6.   .toMat(lastSnk)(Keep.both)
7.   .run()
8.
9. doSomethingElse()
10.
11. killSwitch.shutdown()
12.
13. Await.result(last, 1.second) shouldBe 2
```

- **Abort**

```
1. val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.backpressure)
2. val lastSnk = Sink.last[Int]
3.
4. val (killSwitch, last) = countingSrc
5.   .viaMat(KillSwitches.single)(Keep.right)
6.   .toMat(lastSnk)(Keep.both).run()
```

```
7.
8. val error = new RuntimeException("boom!")
9. killSwitch.abort(error)
10.
11. Await.result(last.failed, 1.second) shouldBe error
```

## SharedKillSwitch

A `SharedKillSwitch` allows to control the completion of an arbitrary number graphs of `FlowShape`. It can be materialized multiple times via its `flow` method, and all materialized graphs linked to it are controlled by the switch. Refer to the below for usage examples.

- **Shutdown**

```
1. val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.backpressure)
2. val lastSnk = Sink.last[Int]
3. val sharedKillSwitch = KillSwitches.shared("my-kill-switch")
4.
5. val last = countingSrc
6.   .via(sharedKillSwitch.flow)
7.   .runWith(lastSnk)
8.
9. val delayedLast = countingSrc
10.   .delay(1.second, DelayOverflowStrategy.backpressure)
11.   .via(sharedKillSwitch.flow)
12.   .runWith(lastSnk)
13.
14. doSomethingElse()
15.
16. sharedKillSwitch.shutdown()
17.
18. Await.result(last, 1.second) shouldBe 2
19. Await.result(delayedLast, 1.second) shouldBe 1
```

- **Abort**

```
1. val countingSrc = Source(Stream.from(1)).delay(1.second)
2. val lastSnk = Sink.last[Int]
3. val sharedKillSwitch = KillSwitches.shared("my-kill-switch")
4.
5. val last1 = countingSrc.via(sharedKillSwitch.flow).runWith(lastSnk)
6. val last2 = countingSrc.via(sharedKillSwitch.flow).runWith(lastSnk)
7.
8. val error = new RuntimeException("boom!")
9. sharedKillSwitch.abort(error)
10.
11. Await.result(last1.failed, 1.second) shouldBe error
12. Await.result(last2.failed, 1.second) shouldBe error
```

**Note**

A `UniqueKillSwitch` is always a result of a materialization, whilst `SharedKillSwitch` needs to be constructed before any materialization takes place.

# Custom stream processing

While the processing vocabulary of Akka Streams is quite rich (see the Streams Cookbook for examples) it is sometimes necessary to define new transformation stages either because some functionality is missing from the stock operations, or for performance reasons. In this part we show how to build custom processing stages and graph junctions of various kinds.

**Note**

A custom graph stage should not be the first tool you reach for, defining graphs using flows and the graph DSL is in general easier and does to a larger extent protect you from mistakes that might be easy to make with a custom `GraphStage`

## Custom processing with GraphStage

The `GraphStage` abstraction can be used to create arbitrary graph processing stages with any number of input or output ports. It is a counterpart of the `GraphDSL.create()` method which creates new stream processing stages by composing others. Where `GraphStage` differs is that it creates a stage that is itself not divisible into smaller ones, and allows state to be maintained inside it in a safe way.

As a first motivating example, we will build a new `Source` that will simply emit numbers from 1 until it is cancelled. To start, we need to define the "interface" of our stage, which is called *shape* in Akka Streams terminology (this is explained in more detail in the section Modularity, Composition and Hierarchy). This is how this looks like:

```
1.  import akka.stream.SourceShape
2.  import akka.stream.stage.GraphStage
3.
4.  class NumbersSource extends GraphStage[SourceShape[Int]] {
5.    // Define the (sole) output port of this stage
6.    val out: Outlet[Int] = Outlet("NumbersSource")
7.    // Define the shape of this stage, which is SourceShape with the port we defined above
8.    override val shape: SourceShape[Int] = SourceShape(out)
9.
10.   // This is where the actual (possibly stateful) logic will live
11.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = ???
12. }
```

As you see, in itself the `GraphStage` only defines the ports of this stage and a shape that contains the ports. It also has, a currently unimplemented method called `createLogic`. If you recall, stages are reusable in multiple materializations, each resulting in a different executing entity. In the case of `GraphStage` the actual running logic is modeled as an instance of a `GraphStageLogic` which will be created by the materializer by calling the `createLogic` method. In other words, all we need to do is to create a suitable logic that will emit the numbers we want.

**Note**

It is very important to keep the GraphStage object itself immutable and reusable. All mutable state needs to be confined to the GraphStageLogic that is created for every materialization.

In order to emit from a `Source` in a backpressured stream one needs first to have demand from downstream. To receive the necessary events one needs to register a subclass of `OutHandler` with the output port (`Outlet`). This handler will receive events related to the lifecycle of the port. In our case we need to

override `onPull()` which indicates that we are free to emit a single element. There is another callback, `onDownstreamFinish()` which is called if the downstream cancelled. Since the default behavior of that callback is to stop the stage, we don't need to override it. In the `onPull` callback we will simply emit the next number. This is how it looks like in the end:

```
1. import akka.stream.SourceShape
2. import akka.stream.Graph
3. import akka.stream.stage.GraphStage
4. import akka.stream.stage.OutHandler
5.
6. class NumbersSource extends GraphStage[SourceShape[Int]] {
7.   val out: Outlet[Int] = Outlet("NumbersSource")
8.   override val shape: SourceShape[Int] = SourceShape(out)
9.
10.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
11.     new GraphStageLogic(shape) {
12.       // All state MUST be inside the GraphStageLogic,
13.       // never inside the enclosing GraphStage.
14.       // This state is safe to access and modify from all the
15.       // callbacks that are provided by GraphStageLogic and the
16.       // registered handlers.
17.       private var counter = 1
18.
19.       setHandler(out, new OutHandler {
20.         override def onPull(): Unit = {
21.           push(out, counter)
22.           counter += 1
23.         }
24.       })
25.     }
26. }
```

Instances of the above `GraphStage` are subclasses of `Graph[SourceShape[Int],Unit]` which means that they are already usable in many situations, but do not provide the DSL methods we usually have for other `Source` s. In order to convert this `Graph` to a proper `Source` we need to wrap it

using `Source.fromGraph` (see Modularity, Composition and Hierarchy for more details about graphs and DSLs). Now we can use the source as any other built-in one:

```
1. // A GraphStage is a proper Graph, just like what GraphDSL.create would return
2. val sourceGraph: Graph[SourceShape[Int], NotUsed] = new NumbersSource
3.
4. // Create a Source from the Graph to access the DSL
5. val mySource: Source[Int, NotUsed] = Source.fromGraph(sourceGraph)
6.
7. // Returns 55
8. val result1: Future[Int] = mySource.take(10).runFold(0)(_ + _)
9.
10. // The source is reusable. This returns 5050
11. val result2: Future[Int] = mySource.take(100).runFold(0)(_ + _)
```

Similarly, to create a custom `Sink` one can register a subclass `InHandler` with the stage `Inlet`. The `onPush()` callback is used to signal the handler a new element has been pushed to the stage, and can hence be grabbed and used. `onPush()` can be overridden to provide custom behaviour. Please note, most Sinks would need to request upstream elements as soon as they are created: this can be done by calling `pull(inlet)` in the `preStart()` callback.

```
1. import akka.stream.SinkShape
2. import akka.stream.stage.GraphStage
3. import akka.stream.stage.InHandler
4.
5. class StdoutSink extends GraphStage[SinkShape[Int]] {
6.   val in: Inlet[Int] = Inlet("StdoutSink")
7.   override val shape: SinkShape[Int] = SinkShape(in)
8.
9.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
10.     new GraphStageLogic(shape) {
11.
12.       // This requests one element at the Sink startup.
13.       override def preStart(): Unit = pull(in)
14.
```

```
15.       setHandler(in, new InHandler {
16.         override def onPush(): Unit = {
17.           println(grab(in))
18.           pull(in)
19.         }
20.       })
21.     }
22. }
```

## Port states, InHandler and OutHandler

In order to interact with a port (`Inlet` or `Outlet`) of the stage we need to be able to receive events and generate new events belonging to the port. From the `GraphStageLogic` the following operations are available on an output port:

- `push(out,elem)` pushes an element to the output port. Only possible after the port has been pulled by downstream.
- `complete(out)` closes the output port normally.
- `fail(out,exception)` closes the port with a failure signal.

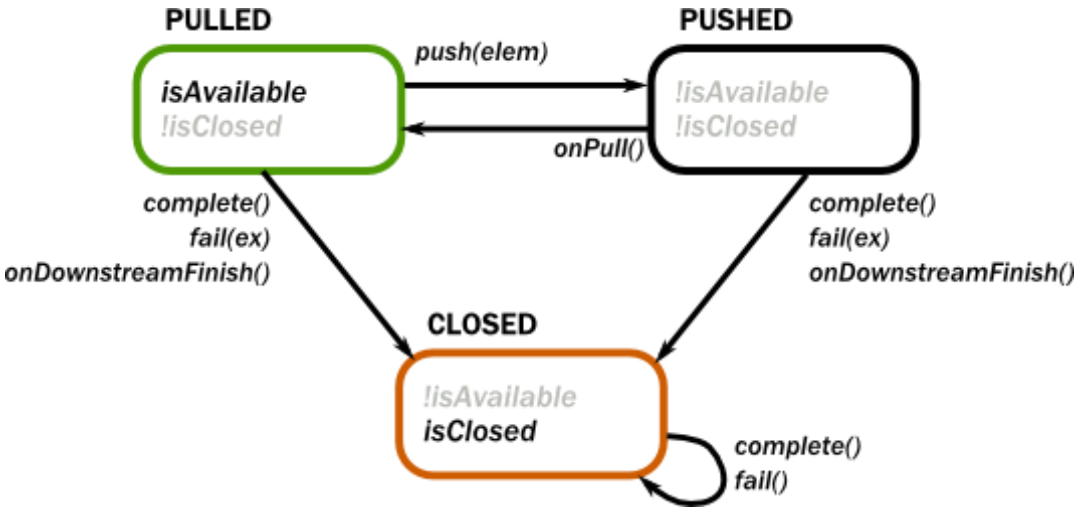
The events corresponding to an *output* port can be received in an `OutHandler` instance registered to the output port using `setHandler(out,handler)`. This handler has two callbacks:

- `onPull()` is called when the output port is ready to emit the next element, `push(out, elem)` is now allowed to be called on this port.
- `onDownstreamFinish()` is called once the downstream has cancelled and no longer allows messages to be pushed to it. No more `onPull()` will arrive after this event. If not overridden this will default to stopping the stage.

Also, there are two query methods available for output ports:

- `isAvailable(out)` returns true if the port can be pushed
- `isClosed(out)` returns true if the port is closed. At this point the port can not be pushed and will not be pulled anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.




The following operations are available for *input* ports:

- `pull(in)` requests a new element from an input port. This is only possible after the port has been pushed by upstream.
- `grab(in)` acquires the element that has been received during an `onPush()`. It cannot be called again until the port is pushed again by the upstream.
- `cancel(in)` closes the input port.

The events corresponding to an *input* port can be received in an `InHandler` instance registered to the input port using `setHandler(in, handler)`. This handler has three callbacks:

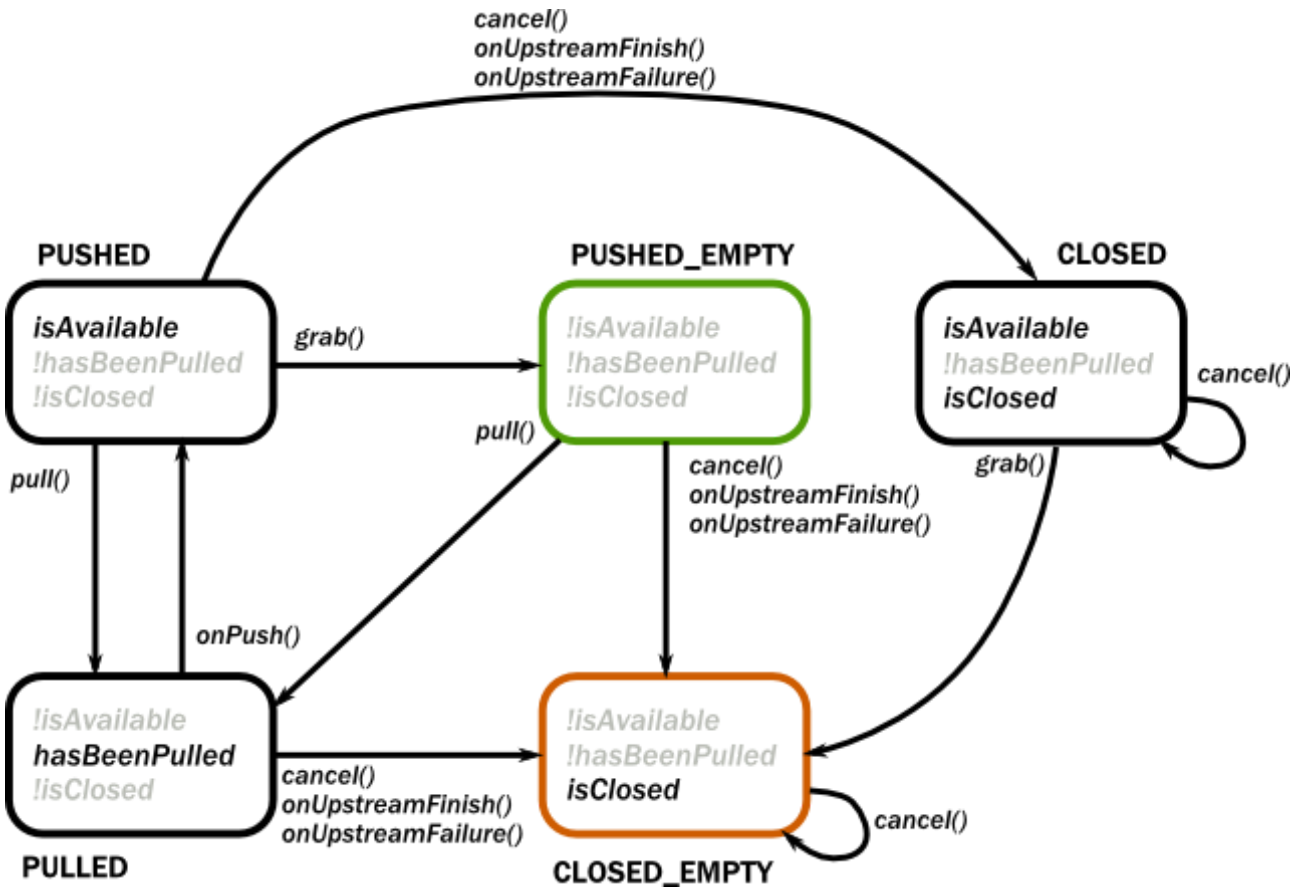
- `onPush()` is called when the input port has now a new element. Now it is possible to acquire this element using `grab(in)` and/or call `pull(in)` on the port to request the next element. It is not mandatory to grab the element, but if it is pulled while the element has not been grabbed it will drop the buffered element.
- `onUpstreamFinish()` is called once the upstream has completed and no longer can be pulled for new elements. No more `onPush()` will arrive after this event. If not overridden this will default to stopping the stage.
- `onUpstreamFailure()` is called if the upstream failed with an exception and no longer can be pulled for new elements. No more `onPush()` will arrive after this event. If not overridden this will default to failing the stage.

Also, there are three query methods available for input ports:

- `isAvailable(in)` returns true if the port can be grabbed.
- `hasBeenPulled(in)` returns true if the port has been already pulled. Calling `pull(in)` in this state is illegal.
- `isClosed(in)` returns true if the port is closed. At this point the port can not be pulled and will not be pushed anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in

that state.




Finally, there are two methods available for convenience to complete the 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.

In some cases it is inconvenient and error prone to react on the regular state machine events with the signal based API described above. For those cases there is an API which allows for a more declarative sequencing of actions which will greatly simplify some use cases at the cost of some extra allocations. The difference between the two APIs could be described as that the first one is signal driven from the outside, while this API is more active and drives its surroundings.

The operations of this part of the :class:`GraphStage` API are:

- `emit(out, elem)` and `emitMultiple(out, Iterable(elem1, elem2))` replaces the `OutHandler` with a handler that emits one or more elements when there is demand, and then reinstalls the current handlers
- `read(in)(andThen)` and `readN(in, n)(andThen)` replaces the `InHandler` with a handler that reads one or more elements as they are pushed and allows the handler to react once the requested number of elements has been read.
- `abortEmitting()` and `abortReading()` which will cancel an ongoing emit or read

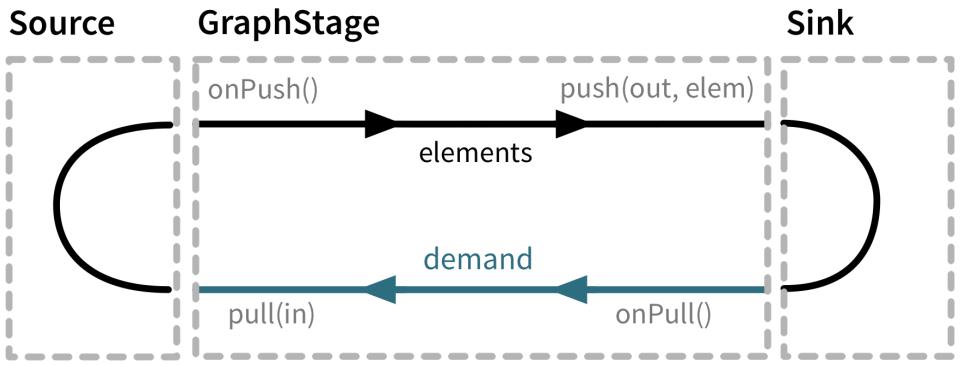
Note that since the above methods are implemented by temporarily replacing the handlers of the stage you should never call `setHandler` while they are running `emit` or `read` as that interferes with how they are implemented. The following methods are safe to call after invoking `emit` and `read` (and will lead to actually running the operation when those are done): `complete(out)`, `completeStage()`, `emit`, `emitMultiple`, `abortEmitting()` and `abortReading()`

An example of how this API simplifies a stage can be found below in the second version of the :class:`Duplicator`.

## Custom linear processing stages using GraphStage

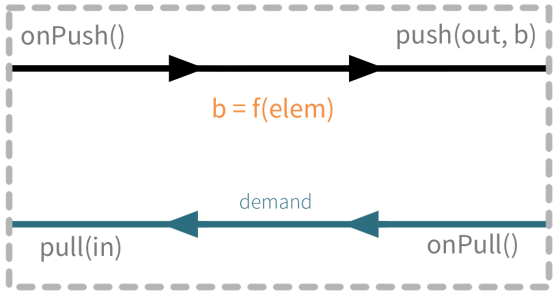
Graph stages allows for custom linear processing stages through letting them have one input and one output and using `FlowShape` as their shape.

Such a stage can be illustrated as a box with two flows as it is seen in the illustration below. Demand flowing upstream leading to elements flowing downstream.




To illustrate these concepts we create a small `GraphStage` that implements the `map` transformation.

**Map**



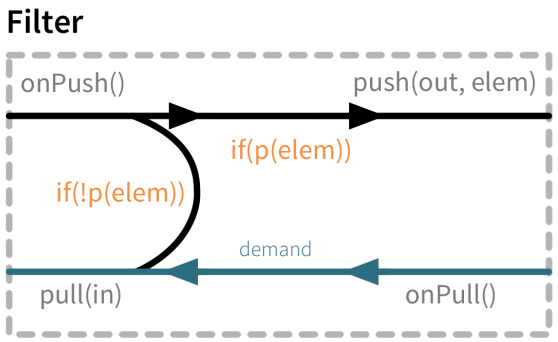

Map calls `push(out)` from the `onPush()` handler and it also calls `pull()` from the `onPull` handler resulting in the conceptual wiring above, and fully expressed in code below:

```
1.  class Map[A, B](f: A => B) extends GraphStage[FlowShape[A, B]] {
2.
3.    val in = Inlet[A]("Map.in")
4.    val out = Outlet[B]("Map.out")
5.
6.    override val shape = FlowShape.of(in, out)
7.
8.    override def createLogic(attr: Attributes): GraphStageLogic =
9.      new GraphStageLogic(shape) {
10.       setHandler(in, new InHandler {
11.         override def onPush(): Unit = {
12.           push(out, f(grab(in)))
13.         }
14.       })
15.       setHandler(out, new OutHandler {
16.         override def onPull(): Unit = {
17.           pull(in)
18.         }
19.       })
```

```
20.     }
21. }
```

Map is a typical example of a one-to-one transformation of a stream where demand is passed along upstream elements passed on downstream.

To demonstrate a many-to-one stage we will implement filter. The conceptual wiring of `Filter` looks like this:

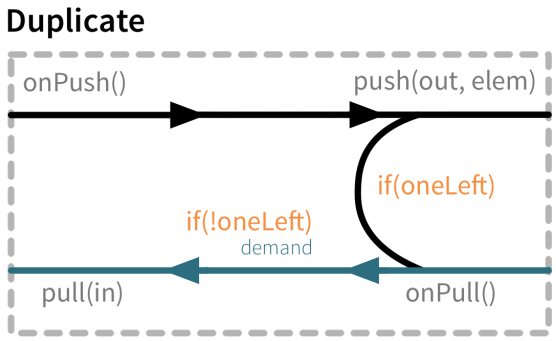



As we see above, if the given predicate matches the current element we are propagating it downwards, otherwise we return the “ball” to our upstream so that we get the new element. This is achieved by modifying the map example by adding a conditional in the `onPush` handler and decide between a `pull(in)` or `push(out)` call (and of course not having a mapping `f` function).

```
1. class Filter[A](p: A => Boolean) extends GraphStage[FlowShape[A, A]] {
2.
3.   val in = Inlet[A]("Filter.in")
4.   val out = Outlet[A]("Filter.out")
5.
6.   val shape = FlowShape.of(in, out)
7.
8.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
9.     new GraphStageLogic(shape) {
10.      setHandler(in, new InHandler {
11.        override def onPush(): Unit = {
```

```
12.           val elem = grab(in)
13.           if (p(elem)) push(out, elem)
14.           else pull(in)
15.         }
16.       })
17.       setHandler(out, new OutHandler {
18.         override def onPull(): Unit = {
19.           pull(in)
20.         }
21.       })
22.     }
23. }
```

To complete the picture we define a one-to-many transformation as the next step. We chose a straightforward example stage that emits every upstream element twice downstream. The conceptual wiring of this stage looks like this:




This is a stage that has state: an option with the last element it has seen indicating if it has duplicated this last element already or not. We must also make sure to emit the extra element if the upstream completes.

```
1. class Duplicator[A] extends GraphStage[FlowShape[A, A]] {
2.
3.   val in = Inlet[A]("Duplicator.in")
```

```
  val out = Outlet[A]("Duplicator.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // Again: note that all mutable state
      // MUST be inside the GraphStageLogic
      var lastElem: Option[A] = None

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          lastElem = Some(elem)
          push(out, elem)
        }

        override def onUpstreamFinish(): Unit = {
          if (lastElem.isDefined) emit(out, lastElem.get)
          complete(out)
        }

      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (lastElem.isDefined) {
            push(out, lastElem.get)
            lastElem = None
          } else {
            pull(in)
          }
        }
      })
```

```
    }
}
```

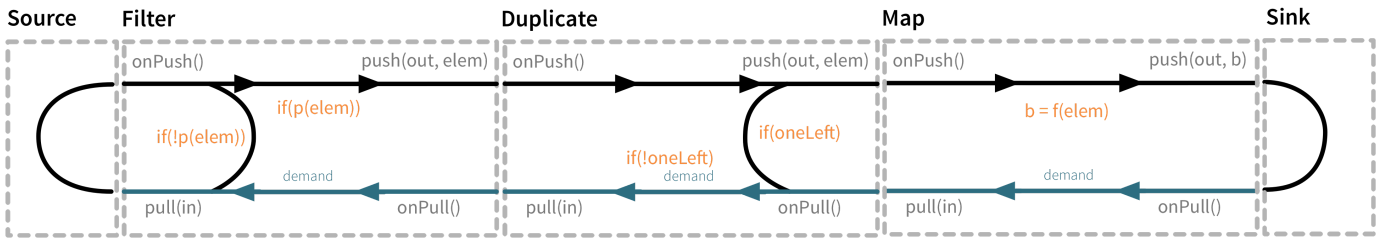
In this case a pull from downstream might be consumed by the stage itself rather than passed along upstream as the stage might contain an element it wants to push. Note that we also need to handle the case where the upstream closes while the stage still has elements it wants to push downstream. This is done by overriding onUpstreamFinish in theInHandler and provide custom logic that should happen when the upstream has been finished.

This example can be simplified by replacing the usage of a mutable state with calls to `emitMultiple` which will replace the handlers, emit each of multiple elements and then reinstate the original handlers:

```
class Duplicator[A] extends GraphStage[FlowShape[A, A]] {

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

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          // this will temporarily suspend this handler until the two elems
          // are emitted and then reinstates it
          emitMultiple(out, Iterable(elem, elem))
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
```

```
25. }
```

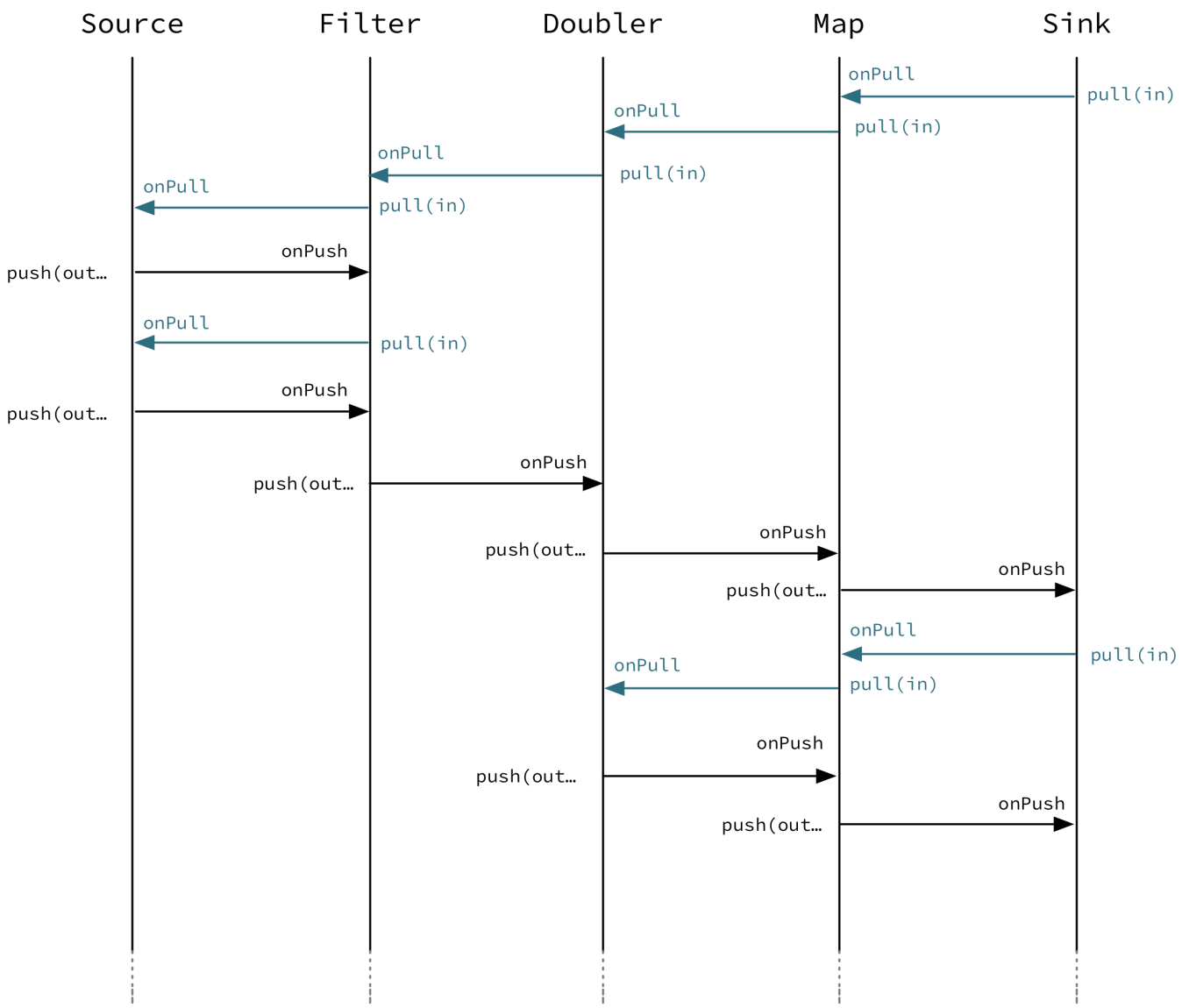
Finally, to demonstrate all of the stages above, we put them together into a processing chain, which conceptually would correspond to the following structure:




In code this is only a few lines, using the `via` use our custom stages in a stream:

```
1. val resultFuture = Source(1 to 5)
2.    .via(new Filter(_ % 2 == 0))
3.    .via(new Duplicator())
4.    .via(new Map(_ / 2))
5.    .runWith(sink)
```

If we attempt to draw the sequence of events, it shows that there is one "event token" in circulation in a potential chain of stages, just like our conceptual "railroad tracks" representation predicts.




## Completion

Completion handling usually (but not exclusively) comes into the picture when processing stages need to emit a few more elements after their upstream source has been completed. We have seen an example of this in our first `Duplicator` implementation where the last element needs to be doubled even after the upstream neighbor stage has been completed. This can be done by overriding the `onUpstreamFinish` method in `InHandler`.

Stages by default automatically stop once all of their ports (input and output) have been closed externally or internally. It is possible to opt out from this behavior by invoking `setKeepGoing(true)` (which is not supported from the stage's constructor and usually done in `preStart`). In this case the stage **must** be explicitly closed by calling `completeStage()` or `failStage(exception)`. This feature carries the risk of leaking streams and actors, therefore it should be used with care.

## Using timers

It is possible to use timers in `GraphStages` by using `TimerGraphStageLogic` as the base class for the returned logic. Timers can be scheduled by calling one of `scheduleOnce(key,delay)`, `schedulePeriodically(key,period)` or `schedulePeriodicallyWithInitialDelay(key,delay,period)` and passing an object as a key for that timer (can be any object, for example a `String`). The `onTimer(key)` method needs to be overridden and it will be called once the timer of `key` fires. It is possible to cancel a timer using `cancelTimer(key)` and check the status of a timer with `isTimerActive(key)`. Timers will be automatically cleaned up when the stage completes.

Timers can not be scheduled from the constructor of the logic, but it is possible to schedule them from the `preStart()` lifecycle hook.

In this sample the stage toggles between open and closed, where open means no elements are passed through. The stage starts out as closed but as soon as an element is pushed downstream the gate becomes open for a duration of time during which it will consume and drop upstream messages:

```
1.  // each time an event is pushed through it will trigger a period of silence
2.  class TimedGate[A](silencePeriod: FiniteDuration) extends GraphStage[FlowShape[A, A]] {
3.
4.    val in = Inlet[A]("TimedGate.in")
5.    val out = Outlet[A]("TimedGate.out")
6.
7.    val shape = FlowShape.of(in, out)
8.
9.    override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
10.     new TimerGraphStageLogic(shape) {
11.
12.       var open = false
13.
14.       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
    }
  }
}
```

## Using asynchronous side-channels

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) one must acquire a `AsyncCallback` by calling `getAsyncCallback()` from the stage logic. The method `getAsyncCallback` takes as a parameter a callback that will be called once the asynchronous event fires. It is important to **not call the callback directly**, instead, the external API must call the `invoke(event)` method on the returned `AsyncCallback`. The execution engine will take care of calling the provided callback in a thread-safe way. The callback can safely access the state of the `GraphStageLogic` implementation.

Sharing the AsyncCallback from the constructor risks race conditions, therefore it is recommended to use the `preStart()` lifecycle hook instead.

This example shows an asynchronous side channel graph stage that starts dropping elements when a future completes:

```
// will close upstream in all materializations of the graph stage instance
// when the future completes
```

```
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) }
      })
    }
}
```

## Integration with actors

**This section is a stub and will be extended in the next release This is an experimental feature***

It is possible to acquire an ActorRef that can be addressed from the outside of the stage, similarly how `AsyncCallback` allows injecting asynchronous events into a stage logic. This reference can be obtained by calling `getStageActorRef(receive)` passing in a function that takes a `Pair` of the sender `ActorRef` and the received message. This reference can be used to watch other actors by calling its `watch(ref)` or `unwatch(ref)` methods. The reference can be also watched by

external actors. The current limitations of this `ActorRef` are:

- they are not location transparent, they cannot be accessed via remoting.
- they cannot be returned as materialized values.
- they cannot be accessed from the constructor of the `GraphStageLogic`, but they can be accessed from the `preStart()` method.

## Custom materialized values

Custom stages can return materialized values instead of `Unit` by inheriting from `GraphStageWithMaterializedValue` instead of the simpler `GraphStage`. The difference is that in this case the method `createLogicAndMaterializedValue(inheritedAttributes)` needs to be overridden, and in addition to the stage logic the materialized value must be provided

**Warning**

There is no built-in synchronization of accessing this value from both of the thread where the logic runs and the thread that got hold of the materialized value. It is the responsibility of the programmer to add the necessary (non-blocking) synchronization and visibility guarantees to this shared object.

In this sample the materialized value is a future containing the first element to go through the stream:

```
1.  class FirstValue[A] extends GraphStageWithMaterializedValue[FlowShape[A, A], Future[A]] {
2.
3.    val in = Inlet[A]("FirstValue.in")
4.    val out = Outlet[A]("FirstValue.out")
5.
6.    val shape = FlowShape.of(in, out)
7.
8.    override def createLogicAndMaterializedValue(inheritedAttributes: Attributes): (GraphStageLogic, Future[A]) = {
9.      val promise = Promise[A]()
10.     val logic = new GraphStageLogic(shape) {
11.
12.       setHandler(in, new InHandler {
13.         override def onPush(): Unit = {
14.           val elem = grab(in)
15.           promise.success(elem)
```

```
16.           push(out, elem)
17.
18.           // replace handler with one just forwarding
19.           setHandler(in, new InHandler {
20.             override def onPush(): Unit = {
21.               push(out, grab(in))
22.             }
23.           })
24.         }
25.       })
26.
27.       setHandler(out, new OutHandler {
28.         override def onPull(): Unit = {
29.           pull(in)
30.         }
31.       })
32.
33.     }
34.
35.     (logic, promise.future)
36.   }
37. }
```

## Using attributes to affect the behavior of a stage

**This section is a stub and will be extended in the next release**

Stages can access the `Attributes` object created by the materializer. This contains all the applied (inherited) attributes applying to the stage, ordered from least specific (outermost) towards the most specific (innermost) attribute. It is the responsibility of the stage to decide how to reconcile this inheritance chain to a final effective decision.

See Modularity, Composition and Hierarchy for an explanation on how attributes work.

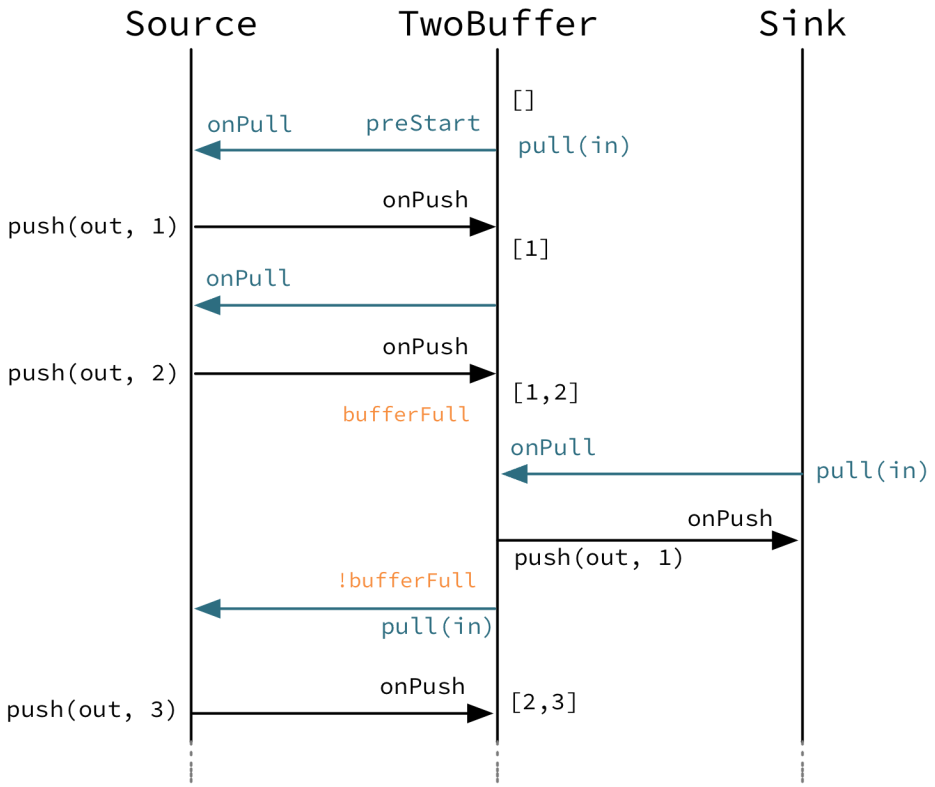
## Rate decoupled graph stages

Sometimes it is desirable to *decouple* the rate of the upstream and downstream of a stage, synchronizing only when needed.

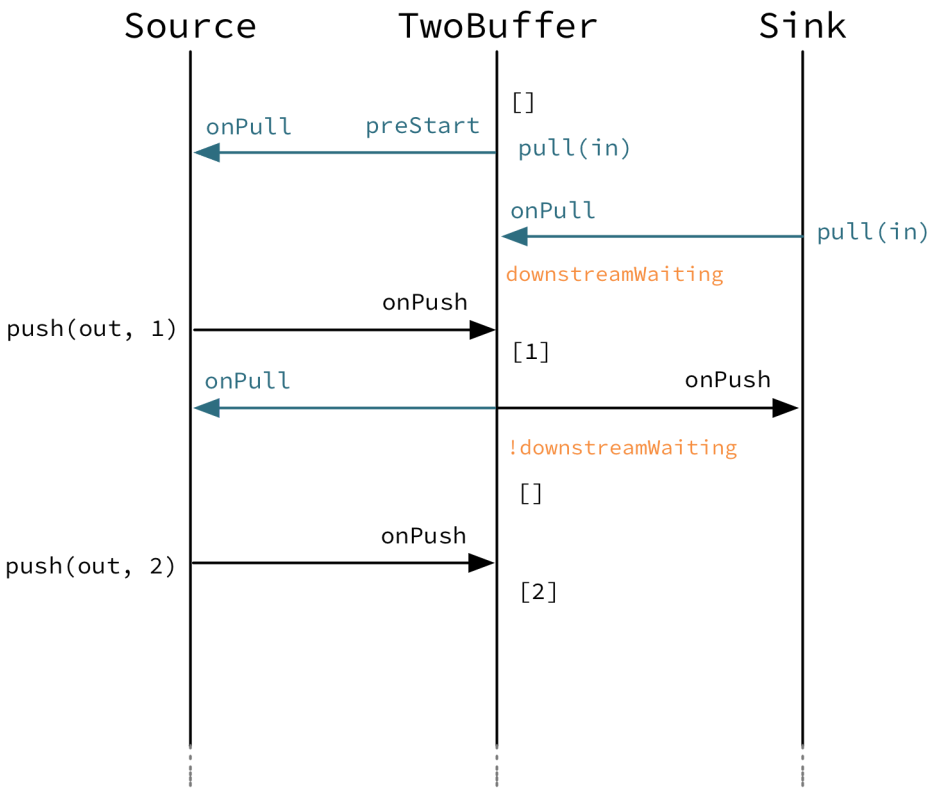
This is achieved in the model by representing a `GraphStage` as a *boundary* between two regions where the demand sent upstream is decoupled from the demand that arrives from downstream. One immediate consequence of this difference is that an `onPush` call does not always lead to calling `push` and an `onPull` call does not always lead to calling `pull`.

One of the important use-case for this is to build buffer-like entities, that allow independent progress of upstream and downstream stages when the buffer is not full or empty, and slowing down the appropriate side if the buffer becomes empty or full.

The next diagram illustrates the event sequence for a buffer with capacity of two elements in a setting where the downstream demand is slow to start and the buffer will fill up with upstream elements before any demand is seen from downstream.



Another scenario would be where the demand from downstream starts coming in before any element is pushed into the buffer stage.




The first difference we can notice is that our `Buffer` stage is automatically pulling its upstream on initialization. The buffer has demand for up to two elements without any downstream demand.

The following code example demonstrates a buffer class corresponding to the message sequence chart above.

```
1. class TwoBuffer[A] extends GraphStage[FlowShape[A, A]] {
2.
3.   val in = Inlet[A]("TwoBuffer.in")
4.   val out = Outlet[A]("TwoBuffer.out")
```

```
5.
6.   val shape = FlowShape.of(in, out)
7.
8.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
9.     new GraphStageLogic(shape) {
10.
11.      val buffer = mutable.Queue[A]()
12.      def bufferFull = buffer.size == 2
13.      var downstreamWaiting = false
14.
15.      override def preStart(): Unit = {
16.        // a detached stage needs to start upstream demand
17.        // itself as it is not triggered by downstream demand
18.        pull(in)
19.      }
20.
21.      setHandler(in, new InHandler {
22.        override def onPush(): Unit = {
23.          val elem = grab(in)
24.          buffer.enqueue(elem)
25.          if (downstreamWaiting) {
26.            downstreamWaiting = false
27.            val bufferedElem = buffer.dequeue()
28.            push(out, bufferedElem)
29.          }
30.          if (!bufferFull) {
31.            pull(in)
32.          }
33.        }
34.
35.        override def onUpstreamFinish(): Unit = {
36.          if (buffer.nonEmpty) {
37.            // emit the rest if possible
```

```
38.             emitMultiple(out, buffer.toIterator)
39.           }
40.           completeStage()
41.         }
42.       })
43.
44.       setHandler(out, new OutHandler {
45.         override def onPull(): Unit = {
46.           if (buffer.isEmpty) {
47.             downstreamWaiting = true
48.           } else {
49.             val elem = buffer.dequeue
50.             push(out, elem)
51.           }
52.           if (!bufferFull && !hasBeenPulled(in)) {
53.             pull(in)
54.           }
55.         }
56.       })
57.     }
58.
59. }
```

## Thread safety of custom processing stages

**All of the above custom stages (linear or graph) provide a few simple guarantees that implementors can rely on.**

- The callbacks exposed by all of these classes are never called concurrently.
- The state encapsulated by these classes can be safely modified from the provided callbacks, without any further synchronization.

In essence, the above guarantees are similar to what `Actor` s provide, if one thinks of the state of a custom stage as state of an actor, and the callbacks as the `receive` block of the actor.

**Warning**

It is **not safe** to access the state of any custom stage outside of the callbacks that it provides, just like it is unsafe to access the state of an actor from the outside. This means that Future callbacks should **not close over** internal state of custom stages because such access can be concurrent with the provided callbacks, leading to undefined behavior.

## Extending Flow Combinators with Custom Operators

The most general way of extending any `Source`, `Flow` or `SubFlow` (e.g. from `groupBy`) is demonstrated above: create a graph of flow-shape like the `Duplicator` example given above and use the `.via(...)` combinator to integrate it into your stream topology. This works with all `FlowOps` sub-types, including the ports that you connect with the graph DSL.

Advanced Scala users may wonder whether it is possible to write extension methods that enrich `FlowOps` to allow nicer syntax. The short answer is that Scala 2 does not support this in a fully generic fashion, the problem is that it is impossible to abstract over the kind of stream that is being extended because `Source`, `Flow` and `SubFlow` differ in the number and kind of their type parameters. While it would be possible to write an implicit class that enriches them generically, this class would require explicit instantiation with all type parameters due to SI-2712. For a partial workaround that unifies extensions to `Source` and `Flow` see this sketch by R. Kuhn.

A lot simpler is the task of just adding an extension method to `Source` as shown below:

```
1. implicit class SourceDuplicator[Out, Mat](s: Source[Out, Mat]) {
2.   def duplicateElements: Source[Out, Mat] = s.via(new Duplicator)
3. }
4.
5. val s = Source(1 to 3).duplicateElements
6.
7. s.runWith(Sink.seq).futureValue should ===(Seq(1, 1, 2, 2, 3, 3))
```

The analog works for `Flow` as well:

```
1. implicit class FlowDuplicator[In, Out, Mat](s: Flow[In, Out, Mat]) {
2.   def duplicateElements: Flow[In, Out, Mat] = s.via(new Duplicator)
```

```
3. }
4.
5. val f = Flow[Int].duplicateElements
6.
7. Source(1 to 3).via(f).runWith(Sink.seq).futureValue should ===(Seq(1, 1, 2, 2, 3, 3))
```

If you try to write this for `SubFlow`, though, you will run into the same issue as when trying to unify the two solutions above, only on a higher level (the type constructors needed for that unification would have rank two, meaning that some of their type arguments are type constructors themselves—when trying to extend the solution shown in the linked sketch the author encountered such a density of compiler StackOverflowErrors and IDE failures that he gave up).

It is interesting to note that a simplified form of this problem has found its way into the dotty test suite. Dotty is the development version of Scala on its way to Scala 3.

# Integration

## Integrating with Actors

For piping the elements of a stream as messages to an ordinary actor you can use the `Sink.actorRef`. Messages can be sent to a stream via the `ActorRef` that is materialized by `Source.actorRef`.

For more advanced use cases the `ActorPublisher` and `ActorSubscriber` traits are provided to support implementing Reactive Streams `Publisher` and `Subscriber` with an `Actor`.

These can be consumed by other Reactive Stream libraries or used as an Akka Streams `Source` or `Sink`.

**Warning**

`ActorPublisher` and `ActorSubscriber` cannot be used with remote actors, because if signals of the Reactive Streams protocol (e.g. `request`) are lost the the stream may deadlock.

### Source.actorRef

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.

Depending on the defined `OverflowStrategy` it might drop elements if there is no space available in the buffer. The strategy `OverflowStrategy.backpressure` is not supported for this Source type, you should consider using `ActorPublisher` if you want a backpressured actor interface.

The stream can be completed successfully by sending `akka.actor.PoisonPill` or `akka.actor.Status.Success` to the actor reference.

The stream can be completed with failure by sending `akka.actor.Status.Failure` to the actor reference.

The actor will be stopped when the stream is completed, failed or cancelled from downstream, i.e. you can watch it to get notified when that happens.

## Sink.actorRef

The sink sends the elements of the stream to the given `ActorRef`. If the target actor terminates the stream will be cancelled. When the stream is completed successfully the given `onCompleteMessage` will be sent to the destination actor. When the stream is completed with failure a `akka.actor.Status.Failure` message will be sent to the destination actor.

**Warning**

There is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow. For potentially slow consumer actors it is recommended to use a bounded mailbox with zero mailbox-push-timeout-time or use a rate limiting stage in front of this stage.

## ActorPublisher

Extend/mixin `akka.stream.actor.ActorPublisher` in your `Actor` to make it a stream publisher that keeps track of the subscription life cycle and requested elements.

Here is an example of such an actor. It dispatches incoming jobs to the attached subscriber:

```
1. object JobManager {
2.   def props: Props = Props[JobManager]
3.
4.   final case class Job(payload: String)
5.   case object JobAccepted
6.   case object JobDenied
7. }
8.
```

```
 9. class JobManager extends ActorPublisher[JobManager.Job] {
10.   import akka.stream.actor.ActorPublisherMessage._
11.   import JobManager._
12.
13.   val MaxBufferSize = 100
14.   var buf = Vector.empty[Job]
15.
16.   def receive = {
17.     case job: Job if buf.size == MaxBufferSize =>
18.       sender() ! JobDenied
19.     case job: Job =>
20.       sender() ! JobAccepted
21.       if (buf.isEmpty && totalDemand > 0)
22.         onNext(job)
23.       else {
24.         buf :+= job
25.         deliverBuf()
26.       }
27.     case Request(_) =>
28.       deliverBuf()
29.     case Cancel =>
30.       context.stop(self)
31.   }
32.
33.   @tailrec final def deliverBuf(): Unit =
34.     if (totalDemand > 0) {
35.       /*
36.        * totalDemand is a Long and could be larger than
37.        * what buf.splitAt can accept
38.        */
39.       if (totalDemand <= Int.MaxValue) {
40.         val (use, keep) = buf.splitAt(totalDemand.toInt)
41.         buf = keep
```

```
        use foreach onNext
      } else {
        val (use, keep) = buf.splitAt(Int.MaxValue)
        buf = keep
        use foreach onNext
        deliverBuf()
      }
    }
}
```

You send elements to the stream by calling `onNext`. You are allowed to send as many elements as have been requested by the stream subscriber. This amount can be inquired with `totalDemand`. It is only allowed to use `onNext` when `isActive` and `totalDemand>0`, otherwise `onNext` will throw `IllegalStateException`.

When the stream subscriber requests more elements the `ActorPublisherMessage.Request` message is delivered to this actor, and you can act on that event. The `totalDemand` is updated automatically.

When the stream subscriber cancels the subscription the `ActorPublisherMessage.Cancel` message is delivered to this actor. After that subsequent calls to `onNext` will be ignored.

You can complete the stream by calling `onComplete`. After that you are not allowed to call `onNext`, `onError` and `onComplete`.

You can terminate the stream with failure by calling `onError`. After that you are not allowed to call `onNext`, `onError` and `onComplete`.

If you suspect that this `ActorPublisher` may never get subscribed to, you can override the `subscriptionTimeout` method to provide a timeout after which this Publisher should be considered canceled. The actor will be notified when the timeout triggers via an `ActorPublisherMessage.SubscriptionTimeoutExceeded` message and MUST then perform cleanup and stop itself.

If the actor is stopped the stream will be completed, unless it was not already terminated with failure, completed or canceled.

More detailed information can be found in the API documentation.

This is how it can be used as input `Source` to a `Flow`:

```
val jobManagerSource = Source.actorPublisher[JobManager.Job](JobManager.props)
val ref = Flow[JobManager.Job]
  .map(_.payload.toUpperCase)
```

```
    .map { elem => println(elem); elem }
    .to(Sink.ignore)
    .runWith(jobManagerSource)

ref ! JobManager.Job("a")
ref ! JobManager.Job("b")
ref ! JobManager.Job("c")
```

A publisher that is created with `Sink.asPublisher` supports a specified number of subscribers. Additional subscription attempts will be rejected with an `IllegalStateException`.

## ActorSubscriber

Extend/mixin `akka.stream.actor.ActorSubscriber` in your `Actor` to make it a stream subscriber with full control of stream back pressure. It will receive `ActorSubscriberMessage.OnNext`, `ActorSubscriberMessage.OnComplete` and `ActorSubscriberMessage.OnError` messages from the stream. It can also receive other, non-stream messages, in the same way as any actor.

Here is an example of such an actor. It dispatches incoming jobs to child worker actors:

```
object WorkerPool {
  case class Msg(id: Int, replyTo: ActorRef)
  case class Work(id: Int)
  case class Reply(id: Int)
  case class Done(id: Int)

  def props: Props = Props(new WorkerPool)
}

class WorkerPool extends ActorSubscriber {
  import WorkerPool._
  import ActorSubscriberMessage._

  val MaxQueueSize = 10
  var queue = Map.empty[Int, ActorRef]
```

```
16.
17.   val router = {
18.     val routees = Vector.fill(3) {
19.       ActorRefRoutee(context.actorOf(Props[Worker]))
20.     }
21.     Router(RoundRobinRoutingLogic(), routees)
22.   }
23.
24.   override val requestStrategy = new MaxInFlightRequestStrategy(max = MaxQueueSize) {
25.     override def inFlightInternally: Int = queue.size
26.   }
27.
28.   def receive = {
29.     case OnNext(Msg(id, replyTo)) =>
30.       queue += (id -> replyTo)
31.       assert(queue.size <= MaxQueueSize, s"queued too many: ${queue.size}")
32.       router.route(Work(id), self)
33.     case Reply(id) =>
34.       queue(id) ! Done(id)
35.       queue -= id
36.   }
37. }
38.
39. class Worker extends Actor {
40.   import WorkerPool._
41.   def receive = {
42.     case Work(id) =>
43.       // ...
44.       sender() ! Reply(id)
45.   }
46. }
```

Subclass must define the `RequestStrategy` to control stream back pressure. After each incoming message the `ActorSubscriber` will automatically invoke the `RequestStrategy.requestDemand` and propagate the returned demand to the stream.

- The provided `WatermarkRequestStrategy` is a good strategy if the actor performs work itself.
- The provided `MaxInFlightRequestStrategy` is useful if messages are queued internally or delegated to other actors.
- You can also implement a custom `RequestStrategy` or call `request` manually together with `ZeroRequestStrategy` or some other strategy. In that case you must also call `request` when the actor is started or when it is ready, otherwise it will not receive any elements.

More detailed information can be found in the API documentation.

This is how it can be used as output `Sink` to a `Flow`:

```
1. val N = 117
2. Source(1 to N).map(WorkerPool.Msg(_, replyTo))
3.   .runWith(Sink.actorSubscriber(WorkerPool.props))
```

## Integrating with External Services

Stream transformations and side effects involving external non-stream based services can be performed with `mapAsync` or `mapAsyncUnordered`.

For example, sending emails to the authors of selected tweets using an external email service:

```
1. def send(email: Email): Future[Unit] = {
2.   // ...
3. }
```

We start with the tweet stream of authors:

```
1. val authors: Source[Author, NotUsed] =
2.   tweets
3.     .filter(_.hashtags.contains(akka))
4.     .map(_.author)
```

Assume that we can lookup their email address using:

```
def lookupEmail(handle: String): Future[Option[String]] =
```

Transforming the stream of authors to a stream of email addresses by using the `lookupEmail` service can be done with `mapAsync`:

```
val emailAddresses: Source[String, NotUsed] =
  authors
    .mapAsync(4)(author => addressSystem.lookupEmail(author.handle))
    .collect { case Some(emailAddress) => emailAddress }
```

Finally, sending the emails:

```
val sendEmails: RunnableGraph[NotUsed] =
  emailAddresses
    .mapAsync(4)(address => {
      emailServer.send(
        Email(to = address, title = "Akka", body = "I like your tweet"))
    })
    .to(Sink.ignore)

sendEmails.run()
```

`mapAsync` is applying the given function that is calling out to the external service to each of the elements as they pass through this processing step. The function returns a `Future` and the value of that future will be emitted downstreams. The number of Futures that shall run in parallel is given as the first argument to `mapAsync`. These Futures may complete in any order, but the elements that are emitted downstream are in the same order as received from upstream.

That means that back-pressure works as expected. For example if the `emailServer.send` is the bottleneck it will limit the rate at which incoming tweets are retrieved and email addresses looked up.

The final piece of this pipeline is to generate the demand that pulls the tweet authors information through the emailing pipeline: we attach a `Sink.ignore` which makes it all run. If our email process would return some interesting data for further transformation then we would of course not ignore it but send that result stream onwards for further processing or storage.

Note that `mapAsync` preserves the order of the stream elements. In this example the order is not important and then we can use the more efficient `mapAsyncUnordered`:

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

val emailAddresses: Source[String, NotUsed] =
  authors
    .mapAsyncUnordered(4)(author => addressSystem.lookupEmail(author.handle))
    .collect { case Some(emailAddress) => emailAddress }

val sendEmails: RunnableGraph[NotUsed] =
  emailAddresses
    .mapAsyncUnordered(4)(address => {
      emailServer.send(
        Email(to = address, title = "Akka", body = "I like your tweet"))
    })
    .to(Sink.ignore)

sendEmails.run()
```

In the above example the services conveniently returned a `Future` of the result. If that is not the case you need to wrap the call in a `Future`. If the service call involves blocking you must also make sure that you run it on a dedicated execution context, to avoid starvation and disturbance of other tasks in the system.

```
val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")

val sendTextMessages: RunnableGraph[NotUsed] =
  phoneNumbers
    .mapAsync(4)(phoneNo => {
      Future {
        smsServer.send(
          TextMessage(to = phoneNo, body = "I like your tweet"))
      }(blockingExecutionContext)
```

```
    })
    .to(Sink.ignore)

sendTextMessages.run()
```

The configuration of the `"blocking-dispatcher"` may look something like:

```
blocking-dispatcher {
  executor = "thread-pool-executor"
  thread-pool-executor {
    core-pool-size-min    = 10
    core-pool-size-max    = 10
  }
}
```

An alternative for blocking calls is to perform them in a `map` operation, still using a dedicated dispatcher for that operation.

```
val send = Flow[String]
  .map { phoneNo =>
    smsServer.send(TextMessage(to = phoneNo, body = "I like your tweet"))
  }
  .withAttributes(ActorAttributes.dispatcher("blocking-dispatcher"))
val sendTextMessages: RunnableGraph[NotUsed] =
  phoneNumbers.via(send).to(Sink.ignore)

sendTextMessages.run()
```

However, that is not exactly the same as `mapAsync`, since the `mapAsync` may run several calls concurrently, but `map` performs them one at a time.

For a service that is exposed as an actor, or if an actor is used as a gateway in front of an external service, you can use `ask`:

```
val akkaTweets: Source[Tweet, NotUsed] = tweets.filter(_.hashtags.contains(akka))
```

```
2.
3. implicit val timeout = Timeout(3.seconds)
4. val saveTweets: RunnableGraph[NotUsed] =
5.   akkaTweets
6.     .mapAsync(4)(tweet => database ? Save(tweet))
7.     .to(Sink.ignore)
```

Note that if the `ask` is not completed within the given timeout the stream is completed with failure. If that is not desired outcome you can use `recover` on the `ask` `Future`.

## Illustrating ordering and parallelism

Let us look at another example to get a better understanding of the ordering and parallelism characteristics of `mapAsync` and `mapAsyncUnordered`.

Several `mapAsync` and `mapAsyncUnordered` futures may run concurrently. The number of concurrent futures are limited by the downstream demand. For example, if 5 elements have been requested by downstream there will be at most 5 futures in progress.

`mapAsync` emits the future results in the same order as the input elements were received. That means that completed results are only emitted downstream when earlier results have been completed and emitted. One slow call will thereby delay the results of all successive calls, even though they are completed before the slow call.

`mapAsyncUnordered` emits the future results as soon as they are completed, i.e. it is possible that the elements are not emitted downstream in the same order as received from upstream. One slow call will thereby not delay the results of faster successive calls as long as there is downstream demand of several elements.

Here is a fictive service that we can use to illustrate these aspects.

```
1. class SometimesSlowService(implicit ec: ExecutionContext) {
2.
3.   private val runningCount = new AtomicInteger
4.
5.   def convert(s: String): Future[String] = {
6.     println(s"running: $s (${runningCount.incrementAndGet()})")
7.     Future {
8.       if (s.nonEmpty && s.head.isLower)
9.         Thread.sleep(500)
```

```
      else
        Thread.sleep(20)
      println(s"completed: $s (${runningCount.decrementAndGet()})")
      s.toUpperCase
    }
  }
}
```

Elements starting with a lower case character are simulated to take longer time to process.

Here is how we can use it with `mapAsync`:

```
implicit val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")
val service = new SometimesSlowService

implicit val materializer = ActorMaterializer(
  ActorMaterializerSettings(system).withInputBuffer(initialSize = 4, maxSize = 4))

Source(List("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem => { println(s"before: $elem"); elem })
  .mapAsync(4)(service.convert)
  .runForeach(elem => println(s"after: $elem"))
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
```

```
before: F
running: D (4)
before: g
before: H
completed: C (3)
completed: B (2)
completed: D (1)
completed: a (0)
after: A
after: B
running: e (1)
after: C
after: D
running: F (2)
before: i
before: J
running: g (3)
running: H (4)
completed: H (2)
completed: F (3)
completed: e (1)
completed: g (0)
after: E
after: F
running: i (1)
after: G
after: H
running: J (2)
completed: J (1)
completed: i (0)
after: I
after: J
```

Note that `after` lines are in the same order as the `before` lines even though elements are `completed` in a different order. For example `H` is `completed` before `g`, but still emitted afterwards.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorMaterializerSettings`.

Here is how we can use the same service with `mapAsyncUnordered`:

```
1. implicit val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")
2. val service = new SometimesSlowService
3.
4. implicit val materializer = ActorMaterializer(
5.   ActorMaterializerSettings(system).withInputBuffer(initialSize = 4, maxSize = 4))
6.
7. Source(List("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
8.   .map(elem => { println(s"before: $elem"); elem })
9.   .mapAsyncUnordered(4)(service.convert)
10.   .runForeach(elem => println(s"after: $elem"))
```

The output may look like this:

```
1. before: a
2. before: B
3. before: C
4. before: D
5. running: a (1)
6. running: B (2)
7. before: e
8. running: C (3)
9. before: F
10. running: D (4)
11. before: g
12. before: H
13. completed: B (3)
```

```
completed: C (1)
completed: D (2)
after: B
after: D
running: e (2)
after: C
running: F (3)
before: i
before: J
completed: F (2)
after: F
running: g (3)
running: H (4)
completed: H (3)
after: H
completed: a (2)
after: A
running: i (3)
running: J (4)
completed: J (3)
after: J
completed: e (2)
after: E
completed: g (1)
after: G
completed: i (0)
after: I
```

Note that `after` lines are not in the same order as the `before` lines. For example `H` overtakes the slow `G`.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorMaterializerSettings`.

# Integrating with Reactive Streams

Reactive Streams defines a standard for asynchronous stream processing with non-blocking back pressure. It makes it possible to plug together stream libraries that adhere to the standard. Akka Streams is one such library.

An incomplete list of other implementations:

- Reactor (1.1+)
- RxJava
- Ratpack
- Slick

The two most important interfaces in Reactive Streams are the `Publisher` and `Subscriber`.

```
1. import org.reactivestreams.Publisher
2. import org.reactivestreams.Subscriber
```

Let us assume that a library provides a publisher of tweets:

```
1. def tweets: Publisher[Tweet]
```

and another library knows how to store author handles in a database:

```
1. def storage: Subscriber[Author]
```

Using an Akka Streams `Flow` we can transform the stream and connect those:

```
1. val authors = Flow[Tweet]
2.   .filter(_.hashtags.contains(akka))
3.   .map(_.author)
4.
5. Source.fromPublisher(tweets).via(authors).to(Sink.fromSubscriber(storage)).run()
```

The `Publisher` is used as an input `Source` to the flow and the `Subscriber` is used as an output `Sink`.

A `Flow` can also be also converted to a `RunnableGraph[Processor[In, Out]]` which materializes to a `Processor` when `run()` is called. `run()` itself can be called multiple times, resulting in a new `Processor` instance each time.

```
1. val processor: Processor[Tweet, Author] = authors.toProcessor.run()
2.
3. tweets.subscribe(processor)
4. processor.subscribe(storage)
```

A publisher can be connected to a subscriber with the `subscribe` method.

It is also possible to expose a `Source` as a `Publisher` by using the Publisher-`Sink`:

```
1. val authorPublisher: Publisher[Author] =
2.   Source.fromPublisher(tweets).via(authors).runWith(Sink.asPublisher(fanout = false))
3.
4. authorPublisher.subscribe(storage)
```

A publisher that is created with `Sink.asPublisher(fanout = false)` supports only a single subscription. Additional subscription attempts will be rejected with an `IllegalStateException`.

A publisher that supports multiple subscribers using fan-out/broadcasting is created as follows:

```
1. def storage: Subscriber[Author]
2. def alert: Subscriber[Author]
```

```
1. val authorPublisher: Publisher[Author] =
2.   Source.fromPublisher(tweets).via(authors)
3.     .runWith(Sink.asPublisher(fanout = true))
4.
5. authorPublisher.subscribe(storage)
```

```
6. authorPublisher.subscribe(alert)
```

The input buffer size of the stage controls how far apart the slowest subscriber can be from the fastest subscriber before slowing down the stream.

To make the picture complete, it is also possible to expose a `Sink` as a `Subscriber` by using the Subscriber-`Source`:

```
1. val tweetSubscriber: Subscriber[Tweet] =
2.   authors.to(Sink.fromSubscriber(storage)).runWith(Source.asSubscriber[Tweet])
3.
4. tweets.subscribe(tweetSubscriber)
```

It is also possible to use re-wrap `Processor` instances as a `Flow` by passing a factory function that will create the `Processor` instances:

```
1. // An example Processor factory
2. def createProcessor: Processor[Int, Int] = Flow[Int].toProcessor.run()
3.
4. val flow: Flow[Int, Int, NotUsed] = Flow.fromProcessor(() => createProcessor)
```

Please note that a factory is necessary to achieve reusability of the resulting `Flow`.

# Error Handling

Strategies for how to handle exceptions from processing stream elements can be defined when materializing the stream. The error handling strategies are inspired by actor supervision strategies, but the semantics have been adapted to the domain of stream processing.

**Warning**

*ZipWith*, *GraphStage* junction, *ActorPublisher* source and *ActorSubscriber* sink components do not honour the supervision strategy attribute yet.

## Supervision Strategies

There are three ways to handle exceptions from application code:

- `Stop` - The stream is completed with failure.
- `Resume` - The element is dropped and the stream continues.
- `Restart` - The element is dropped and the stream continues after restarting the stage. Restarting a stage means that any accumulated state is cleared. This is typically performed by creating a new instance of the stage.

By default the stopping strategy is used for all exceptions, i.e. the stream will be completed with failure when an exception is thrown.

```
1. implicit val materializer = ActorMaterializer()
2. val source = Source(0 to 5).map(100 / _)
3. val result = source.runWith(Sink.fold(0)(_ + _))
4. // division by zero will fail the stream and the
5. // result here will be a Future completed with Failure(ArithmeticException)
```

The default supervision strategy for a stream can be defined on the settings of the materializer.

```
1. val decider: Supervision.Decider = {
2.   case _: ArithmeticException => Supervision.Resume
3.   case _                      => Supervision.Stop
4. }
5. implicit val materializer = ActorMaterializer(
6.   ActorMaterializerSettings(system).withSupervisionStrategy(decider))
7. val source = Source(0 to 5).map(100 / _)
8. val result = source.runWith(Sink.fold(0)(_ + _))
9. // the element causing division by zero will be dropped
10. // result here will be a Future completed with Success(228)
```

Here you can see that all `ArithmeticException` will resume the processing, i.e. the elements that cause the division by zero are effectively dropped.

**Note**

Be aware that dropping elements may result in deadlocks in graphs with cycles, as explained in Graph cycles, liveness and deadlocks.

The supervision strategy can also be defined for all operators of a flow.

```
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)(_ + _))
// the elements causing division by zero will be dropped
// result here will be a Future completed with Success(150)
```

`Restart` works in a similar way as `Resume` with the addition that accumulated state, if any, of the failing processing stage will be reset.

```
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.limit(1000).runWith(Sink.seq)
// the negative element cause the scan stage to be restarted,
// i.e. start from 0 again
```

```
16. // result here will be a Future completed with Success(Vector(0, 1, 4, 0, 5, 12))
```

## Errors from mapAsync

Stream supervision can also be applied to the futures of `mapAsync`.

Let's say that we use an external service to lookup email addresses and we would like to discard those that cannot be found.

We start with the tweet stream of authors:

```
1. val authors: Source[Author, NotUsed] =
2.   tweets
3.     .filter(_.hashtags.contains(akka))
4.     .map(_.author)
```

Assume that we can lookup their email address using:

```
1. def lookupEmail(handle: String): Future[String] =
```

The `Future` is completed with `Failure` if the email is not found.

Transforming the stream of authors to a stream of email addresses by using the `lookupEmail` service can be done with `mapAsync` and we use `Supervision.resumingDecider` to drop unknown email addresses:

```
1. import ActorAttributes.supervisionStrategy
2. import Supervision.resumingDecider
3.
4. val emailAddresses: Source[String, NotUsed] =
5.   authors.via(
6.     Flow[Author].mapAsync(4)(author => addressSystem.lookupEmail(author.handle))
7.       .withAttributes(supervisionStrategy(resumingDecider)))
```

If we would not use `Resume` the default stopping strategy would complete the stream with failure on the first `Future` that was completed with `Failure`.

# Working with streaming IO

Akka Streams provides a way of handling File IO and TCP connections with Streams. While the general approach is very similar to the Actor based TCP handling using Akka IO, by using Akka Streams you are freed of having to manually react to back-pressure signals, as the library does it transparently for you.

## Streaming TCP

### Accepting connections: Echo Server

In order to implement a simple EchoServer we `bind` to a given address, which returns a `Source[IncomingConnection, Future[ServerBinding]]`, which will emit an `IncomingConnection` element for each new connection that the Server should handle:

```
1. val binding: Future[ServerBinding] =
2.   Tcp().bind("127.0.0.1", 8888).to(Sink.ignore).run()
3.
4. binding.map { b =>
5.   b.unbind() onComplete {
6.     case _ => // ...
7.   }
8. }
```

Next, we simply handle *each* incoming connection using a `Flow` which will be used as the processing stage to handle and emit ByteStrings from and to the TCP Socket. Since one `ByteString` does not have to necessarily correspond to exactly one line of text (the client might be sending the line in chunks) we use the `Framing.delimiter` helper Flow to chunk the inputs up into actual lines of text. The last boolean argument indicates that we require an explicit line ending even for the last message before the connection is closed. In this example we simply add exclamation marks to each incoming text message and push it through the flow:

```
1. import akka.stream.scaladsl.Framing
2.
3. val connections: Source[IncomingConnection, Future[ServerBinding]] =
4.   Tcp().bind(host, port)
5. connections runForeach { connection =>
```

```
  println(s"New connection from: ${connection.remoteAddress}")

  val echo = Flow[ByteString]
    .via(Framing.delimiter(
      ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true))
    .map(_.utf8String)
    .map(_ + "!!!\n")
    .map(ByteString(_))

  connection.handleWith(echo)
}
```

Notice that while most building blocks in Akka Streams are reusable and freely shareable, this is *not* the case for the incoming connection Flow, since it directly corresponds to an existing, already accepted connection its handling can only ever be materialized *once*.

Closing connections is possible by cancelling the *incoming connection* `Flow` from your server logic (e.g. by connecting its downstream to a `Sink.cancelled` and its upstream to a `Source.empty`). It is also possible to shut down the server's socket by cancelling the `IncomingConnection` source `connections`.

We can then test the TCP server by sending data to the TCP Socket using `netcat`:

```
$ echo -n "Hello World" | netcat 127.0.0.1 8888
Hello World!!!
```

## Connecting: REPL Client

In this example we implement a rather naive Read Evaluate Print Loop client over TCP. Let's say we know a server has exposed a simple command line interface over TCP, and would like to interact with it using Akka Streams over TCP. To open an outgoing connection socket we use the `outgoingConnection` method:

```
val connection = Tcp().outgoingConnection("127.0.0.1", 8888)

val replParser =
  Flow[String].takeWhile(_ != "q")
```

```
    .concat(Source.single("BYE"))
    .map(elem => ByteString(s"$elem\n"))

val repl = Flow[ByteString]
  .via(Framing.delimiter(
    ByteString("\n"),
    maximumFrameLength = 256,
    allowTruncation = true))
  .map(_.utf8String)
  .map(text => println("Server: " + text))
  .map(_ => readLine("> "))
  .via(replParser)

connection.join(repl).run()
```

The `repl` flow we use to handle the server interaction first prints the servers response, then awaits on input from the command line (this blocking call is used here just for the sake of simplicity) and converts it to a `ByteString` which is then sent over the wire to the server. Then we simply connect the TCP pipeline to this processing stage–at this point it will be materialized and start processing data once the server responds with an *initial message*.

A resilient REPL client would be more sophisticated than this, for example it should split out the input reading into a separate mapAsync step and have a way to let the server write more data than one ByteString chunk at any given time, these improvements however are left as exercise for the reader.

## Avoiding deadlocks and liveness issues in back-pressured cycles

When writing such end-to-end back-pressured systems you may sometimes end up in a situation of a loop, in which*either side is waiting for the other one to start the conversation*. One does not need to look far to find examples of such back-pressure loops. In the two examples shown previously, we always assumed that the side we are connecting to would start the conversation, which effectively means both sides are back-pressured and can not get the conversation started. There are multiple ways of dealing with this which are explained in depth in Graph cycles, liveness and deadlocks, however in client-server scenarios it is often the simplest to make either side simply send an initial message.

**Note**

In case of back-pressured cycles (which can occur even between different systems) sometimes you have to decide which of the sides has start the conversation in order to kick it off. This can be often done by injecting an initial message from one of the sides–a conversation starter.

To break this back-pressure cycle we need to inject some initial message, a "conversation starter". First, we need to decide which side of the connection should remain passive and which active. Thankfully in most situations finding the right spot to start the conversation is rather simple, as it often is inherent to the protocol we are trying to implement using Streams. In chat-like applications, which our examples resemble, it makes sense to make the Server initiate the conversation by emitting a "hello" message:

```
connections.runForeach { connection =>

  // server logic, parses incoming commands
  val commandParser = Flow[String].takeWhile(_ != "BYE").map(_ + "!")

  import connection._
  val welcomeMsg = s"Welcome to: $localAddress, you are: $remoteAddress!"
  val welcome = Source.single(welcomeMsg)

  val serverLogic = Flow[ByteString]
    .via(Framing.delimiter(
      ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true))
    .map(_.utf8String)
    .via(commandParser)
    // merge in the initial banner after parser
    .merge(welcome)
    .map(_ + "\n")
    .map(ByteString(_))

  connection.handleWith(serverLogic)
}
```

To emit the initial message we merge a `Source` with a single element, after the command processing but before the framing and transformation to `ByteStrings` this way we do not have to repeat such logic.

In this example both client and server may need to close the stream based on a parsed command - `BYE` in the case of the server, and `q` in the case of the client. This is implemented by taking from the stream until `q` and and concatenating a `Source` with a single `BYE` element which will then be sent after the original source

completed.

## Streaming File IO

Akka Streams provide simple Sources and Sinks that can work with `ByteString` instances to perform IO operations on files.

Streaming data from a file is as easy as creating a FileIO.fromPath given a target path, and an optional `chunkSize` which determines the buffer size determined as one "element" in such stream:

```
1. import akka.stream.scaladsl._
2. val file = Paths.get("example.csv")
3.
4. val foreach: Future[IOResult] = FileIO.fromPath(file)
5.   .to(Sink.ignore)
6.   .run()
```

Please note that these processing stages are backed by Actors and by default are configured to run on a pre-configured threadpool-backed dispatcher dedicated for File IO. This is very important as it isolates the blocking file IO operations from the rest of the ActorSystem allowing each dispatcher to be utilised in the most efficient way. If you want to configure a custom dispatcher for file IO operations globally, you can do so by changing the `akka.stream.blocking-io-dispatcher`, or for a specific stage by specifying a custom Dispatcher in code, like this:

```
1. FileIO.fromPath(file)
2.   .withAttributes(ActorAttributes.dispatcher("custom-blocking-io-dispatcher"))
```

# Pipelining and Parallelism

Akka Streams processing stages (be it simple operators on Flows and Sources or graph junctions) are "fused" together and executed sequentially by default. This avoids the overhead of events crossing asynchronous boundaries but limits the flow to execute at most one stage at any given time.

In many cases it is useful to be able to concurrently execute the stages of a flow, this is done by explicitly marking them as asynchronous using the `async` method. Each processing stage marked as asynchronous will run in a dedicated actor internally, while all stages not marked asynchronous will run in one single actor.

We will illustrate through the example of pancake cooking how streams can be used for various processing patterns, exploiting the available parallelism on modern computers. The setting is the following: both Patrik and Roland like to make pancakes, but they need to produce sufficient amount in a cooking session to make all of

the children happy. To increase their pancake production throughput they use two frying pans. How they organize their pancake processing is markedly different.

## Pipelining

Roland uses the two frying pans in an asymmetric fashion. The first pan is only used to fry one side of the pancake then the half-finished pancake is flipped into the second pan for the finishing fry on the other side. Once the first frying pan becomes available it gets a new scoop of batter. As an effect, most of the time there are two pancakes being cooked at the same time, one being cooked on its first side and the second being cooked to completion. This is how this setup would look like implemented as a stream:

```
1. // Takes a scoop of batter and creates a pancake with one side cooked
2. val fryingPan1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] =
3.   Flow[ScoopOfBatter].map { batter => HalfCookedPancake() }
4.
5. // Finishes a half-cooked pancake
6. val fryingPan2: Flow[HalfCookedPancake, Pancake, NotUsed] =
7.   Flow[HalfCookedPancake].map { halfCooked => Pancake() }
8.
9. // With the two frying pans we can fully cook pancakes
10. val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
11.   Flow[ScoopOfBatter].via(fryingPan1.async).via(fryingPan2.async)
```

The two `map` stages in sequence (encapsulated in the "frying pan" flows) will be executed in a pipelined way, basically doing the same as Roland with his frying pans:

1. A `ScoopOfBatter` enters `fryingPan1`
2. `fryingPan1` emits a HalfCookedPancake once `fryingPan2` becomes available
3. `fryingPan2` takes the HalfCookedPancake
4. at this point fryingPan1 already takes the next scoop, without waiting for fryingPan2 to finish

The benefit of pipelining is that it can be applied to any sequence of processing steps that are otherwise not parallelisable (for example because the result of a processing step depends on all the information from the previous step). One drawback is that if the processing times of the stages are very different then some of the stages will not be able to operate at full throughput because they will wait on a previous or subsequent stage most of the time. In the pancake example frying the second half of the pancake is usually faster than frying the first half, `fryingPan2` will not be able to operate at full capacity [1].

**Note**

Asynchronous stream processing stages have internal buffers to make communication between them more efficient. For more details about the behavior of these and how to add additional buffers refer to Buffers and working with rate.

## Parallel processing

Patrik uses the two frying pans symmetrically. He uses both pans to fully fry a pancake on both sides, then puts the results on a shared plate. Whenever a pan becomes empty, he takes the next scoop from the shared bowl of batter. In essence he parallelizes the same process over multiple pans. This is how this setup will look like if implemented using streams:

```
1. val fryingPan: Flow[ScoopOfBatter, Pancake, NotUsed] =
2.   Flow[ScoopOfBatter].map { batter => Pancake() }
3.
4. val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] = Flow.fromGraph(GraphDSL.create() { implicit builder =>
5.   val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
6.   val mergePancakes = builder.add(Merge[Pancake](2))
7.
8.   // Using two frying pans in parallel, both fully cooking a pancake from the batter.
9.   // We always put the next scoop of batter to the first frying pan that becomes available.
10.  dispatchBatter.out(0) ~> fryingPan.async ~> mergePancakes.in(0)
11.  // Notice that we used the "fryingPan" flow without importing it via builder.add().
12.  // Flows used this way are auto-imported, which in this case means that the two
13.  // uses of "fryingPan" mean actually different stages in the graph.
14.  dispatchBatter.out(1) ~> fryingPan.async ~> mergePancakes.in(1)
15.
16.  FlowShape(dispatchBatter.in, mergePancakes.out)
17. })
```

The benefit of parallelizing is that it is easy to scale. In the pancake example it is easy to add a third frying pan with Patrik's method, but Roland cannot add a third frying pan, since that would require a third processing step, which is not practically possible in the case of frying pancakes.

One drawback of the example code above that it does not preserve the ordering of pancakes. This might be a problem if children like to track their "own" pancakes. In those cases the `Balance` and `Merge` stages should be replaced by strict-round robing balancing and merging stages that put in and take out pancakes in a strict

order.

A more detailed example of creating a worker pool can be found in the cookbook: Balancing jobs to a fixed pool of workers

## Combining pipelining and parallel processing

The two concurrency patterns that we demonstrated as means to increase throughput are not exclusive. In fact, it is rather simple to combine the two approaches and streams provide a nice unifying language to express and compose them.

First, let's look at how we can parallelize pipelined processing stages. In the case of pancakes this means that we will employ two chefs, each working using Roland's pipelining method, but we use the two chefs in parallel, just like Patrik used the two frying pans. This is how it looks like if expressed as streams:

```
1. val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
2.   Flow.fromGraph(GraphDSL.create() { implicit builder =>
3.
4.     val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
5.     val mergePancakes = builder.add(Merge[Pancake](2))
6.
7.     // Using two pipelines, having two frying pans each, in total using
8.     // four frying pans
9.     dispatchBatter.out(0) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.in(0)
10.    dispatchBatter.out(1) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.in(1)
11.
12.    FlowShape(dispatchBatter.in, mergePancakes.out)
13.  })
```

The above pattern works well if there are many independent jobs that do not depend on the results of each other, but the jobs themselves need multiple processing steps where each step builds on the result of the previous one. In our case individual pancakes do not depend on each other, they can be cooked in parallel, on the other hand it is not possible to fry both sides of the same pancake at the same time, so the two sides have to be fried in sequence.

It is also possible to organize parallelized stages into pipelines. This would mean employing four chefs:

- the first two chefs prepare half-cooked pancakes from batter, in parallel, then putting those on a large enough flat surface.
- the second two chefs take these and fry their other side in their own pans, then they put the pancakes on a shared plate.

This is again straightforward to implement with the streams API:

```
1. val pancakeChefs1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] =
2.   Flow.fromGraph(GraphDSL.create() { implicit builder =>
3.     val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
4.     val mergeHalfPancakes = builder.add(Merge[HalfCookedPancake](2))
5.
6.     // Two chefs work with one frying pan for each, half-frying the pancakes then putting
7.     // them into a common pool
8.     dispatchBatter.out(0) ~> fryingPan1.async ~> mergeHalfPancakes.in(0)
9.     dispatchBatter.out(1) ~> fryingPan1.async ~> mergeHalfPancakes.in(1)
10.
11.    FlowShape(dispatchBatter.in, mergeHalfPancakes.out)
12.  })
13.
14. val pancakeChefs2: Flow[HalfCookedPancake, Pancake, NotUsed] =
15.   Flow.fromGraph(GraphDSL.create() { implicit builder =>
16.     val dispatchHalfPancakes = builder.add(Balance[HalfCookedPancake](2))
17.     val mergePancakes = builder.add(Merge[Pancake](2))
18.
19.     // Two chefs work with one frying pan for each, finishing the pancakes then putting
20.     // them into a common pool
21.     dispatchHalfPancakes.out(0) ~> fryingPan2.async ~> mergePancakes.in(0)
22.     dispatchHalfPancakes.out(1) ~> fryingPan2.async ~> mergePancakes.in(1)
23.
24.     FlowShape(dispatchHalfPancakes.in, mergePancakes.out)
25.   })
26.
27. val kitchen: Flow[ScoopOfBatter, Pancake, NotUsed] = pancakeChefs1.via(pancakeChefs2)
```

This usage pattern is less common but might be usable if a certain step in the pipeline might take wildly different times to finish different jobs. The reason is that there are more balance-merge steps in this pattern compared to the parallel pipelines. This pattern rebalances after each step, while the previous pattern only balances at the entry point of the pipeline. This only matters however if the processing time distribution has a large deviation.

[1] Roland's reason for this seemingly suboptimal procedure is that he prefers the temperature of the second pan to be slightly lower than the first in order to achieve a more homogeneous result.

# Testing streams

Verifying behaviour of Akka Stream sources, flows and sinks can be done using various code patterns and libraries. Here we will discuss testing these elements using:

- simple sources, sinks and flows;
- sources and sinks in combination with `TestProbe` from the `akka-testkit` module;
- sources and sinks specifically crafted for writing tests from the `akka-stream-testkit` module.

It is important to keep your data processing pipeline as separate sources, flows and sinks. This makes them easily testable by wiring them up to other sources or sinks, or some test harnesses that `akka-testkit` or `akka-stream-testkit` provide.

## Built in sources, sinks and combinators

Testing a custom sink can be as simple as attaching a source that emits elements from a predefined collection, running a constructed test flow and asserting on the results that sink produced. Here is an example of a test for a sink:

```
1. val sinkUnderTest = Flow[Int].map(_ * 2).toMat(Sink.fold(0)(_ + _))(Keep.right)
2.
3. val future = Source(1 to 4).runWith(sinkUnderTest)
4. val result = Await.result(future, 100.millis)
5. assert(result == 20)
```

The same strategy can be applied for sources as well. In the next example we have a source that produces an infinite stream of elements. Such source can be tested by asserting that first arbitrary number of elements hold some condition. Here the `grouped` combinator and `Sink.head` are very useful.

```
1. import system.dispatcher
2. import akka.pattern.pipe
3.
4. val sourceUnderTest = Source.repeat(1).map(_ * 2)
5.
6. val future = sourceUnderTest.grouped(10).runWith(Sink.head)
```

```
val result = Await.result(future, 100.millis)
assert(result == Seq.fill(10)(2))
```

When testing a flow we need to attach a source and a sink. As both stream ends are under our control, we can choose sources that tests various edge cases of the flow and sinks that ease assertions.

```
val flowUnderTest = Flow[Int].takeWhile(_ < 5)

val future = Source(1 to 10).via(flowUnderTest).runWith(Sink.fold(Seq.empty[Int])(_ :+ _))
val result = Await.result(future, 100.millis)
assert(result == (1 to 4))
```

## TestKit

Akka Stream offers integration with Actors out of the box. This support can be used for writing stream tests that use familiar `TestProbe` from the `akka-testkit` API.

One of the more straightforward tests would be to materialize stream to a `Future` and then use `pipe` pattern to pipe the result of that future to the probe.

```
import system.dispatcher
import akka.pattern.pipe

val sourceUnderTest = Source(1 to 4).grouped(2)

val probe = TestProbe()
sourceUnderTest.grouped(2).runWith(Sink.head).pipeTo(probe.ref)
probe.expectMsg(100.millis, Seq(Seq(1, 2), Seq(3, 4)))
```

Instead of materializing to a future, we can use a `Sink.actorRef` that sends all incoming elements to the given `ActorRef`. Now we can use assertion methods on `TestProbe` and expect elements one by one as they arrive. We can also assert stream completion by expecting for `onCompleteMessage` which was given to `Sink.actorRef`.

```
case object Tick
```

```
val sourceUnderTest = Source.tick(0.seconds, 200.millis, Tick)

val probe = TestProbe()
val cancellable = sourceUnderTest.to(Sink.actorRef(probe.ref, "completed")).run()

probe.expectMsg(1.second, Tick)
probe.expectNoMsg(100.millis)
probe.expectMsg(200.millis, Tick)
cancellable.cancel()
probe.expectMsg(200.millis, "completed")
```

Similarly to `Sink.actorRef` that provides control over received elements, we can use `Source.actorRef` and have full control over elements to be sent.

```
val sinkUnderTest = Flow[Int].map(_.toString).toMat(Sink.fold("")(_ + _))(Keep.right)

val (ref, future) = Source.actorRef(8, OverflowStrategy.fail)
  .toMat(sinkUnderTest)(Keep.both).run()

ref ! 1
ref ! 2
ref ! 3
ref ! akka.actor.Status.Success("done")

val result = Await.result(future, 100.millis)
assert(result == "123")
```

## Streams TestKit

You may have noticed various code patterns that emerge when testing stream pipelines. Akka Stream has a separate `akka-stream-testkit` module that provides tools specifically for writing stream tests. This module comes with two main components that are `TestSource` and `TestSink` which provide sources and sinks that materialize to probes that allow fluent API.

**Note**

Be sure to add the module `akka-stream-testkit` to your dependencies.

A sink returned by `TestSink.probe` allows manual control over demand and assertions over elements coming downstream.

```
1. val sourceUnderTest = Source(1 to 4).filter(_ % 2 == 0).map(_ * 2)
2.
3. sourceUnderTest
4.   .runWith(TestSink.probe[Int])
5.   .request(2)
6.   .expectNext(4, 8)
7.   .expectComplete()
```

A source returned by `TestSource.probe` can be used for asserting demand or controlling when stream is completed or ended with an error.

```
1. val sinkUnderTest = Sink.cancelled
2.
3. TestSource.probe[Int]
4.   .toMat(sinkUnderTest)(Keep.left)
5.   .run()
6.   .expectCancellation()
```

You can also inject exceptions and test sink behaviour on error conditions.

```
1. val sinkUnderTest = Sink.head[Int]
2.
3. val (probe, future) = TestSource.probe[Int]
4.   .toMat(sinkUnderTest)(Keep.both)
5.   .run()
6. probe.sendError(new Exception("boom"))
```

```
7.
8. Await.ready(future, 100.millis)
9. val Failure(exception) = future.value.get
10. assert(exception.getMessage == "boom")
```

Test source and sink can be used together in combination when testing flows.

```
1. val flowUnderTest = Flow[Int].mapAsyncUnordered(2) { sleep =>
2.   pattern.after(10.millis * sleep, using = system.scheduler)(Future.successful(sleep))
3. }
4.
5. val (pub, sub) = TestSource.probe[Int]
6.   .via(flowUnderTest)
7.   .toMat(TestSink.probe[Int])(Keep.both)
8.   .run()
9.
10. sub.request(n = 3)
11. pub.sendNext(3)
12. pub.sendNext(2)
13. pub.sendNext(1)
14. sub.expectNextUnordered(1, 2, 3)
15.
16. pub.sendError(new Exception("Power surge in the linear subroutine C-47!"))
17. val ex = sub.expectError()
18. assert(ex.getMessage.contains("C-47"))
```

## Fuzzing Mode

For testing, it is possible to enable a special stream execution mode that exercises concurrent execution paths more aggressively (at the cost of reduced performance) and therefore helps exposing race conditions in tests. To enable this setting add the following line to your configuration:

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

**Warning**

Never use this setting in production or benchmarks. This is a testing tool to provide more coverage of your code during tests, but it reduces the throughput of streams. A warning message will be logged if you have this setting enabled.

# Overview of built-in stages and their semantics

## Source stages

These built-in sources are available from `akka.stream.scaladsl.Source`:

### fromIterator

Stream the values from an `Iterator`, requesting the next value when there is demand. The iterator will be created anew for each materialization, which is the reason the method takes a function rather than an iterator directly.

If the iterator perform blocking operations, make sure to run it on a separate dispatcher.

**emits** the next value returned from the iterator

**completes** when the iterator reaches its end

### apply

Stream the values of an `immutable.Seq`.

**emits** the next value of the seq

**completes** when the last element of the seq has been emitted

### single

Stream a single object

**emits** the value once

**completes** when the single value has been emitted

## repeat

Stream a single object repeatedly

**emits** the same value repeatedly when there is demand

**completes** never

## cycle

Stream iterator in cycled manner. Internally new iterator is being created to cycle the one provided via argument meaning when original iterator runs out of elements process will start all over again from the beginning of the iterator provided by the evaluation of provided parameter. If method argument provides empty iterator stream will be terminated with exception.

**emits** the next value returned from cycled iterator

**completes** never

## tick

A periodical repetition of an arbitrary object. Delay of first tick is specified separately from interval of the following ticks.

**emits** periodically, if there is downstream backpressure ticks are skipped

**completes** never

## fromFuture

Send the single value of the `Future` when it completes and there is demand. If the future fails the stream is failed with that exception.

**emits** the future completes

**completes** after the future has completed

## fromCompletionStage

Send the single value of the Java `CompletionStage` when it completes and there is demand. If the future fails the stream is failed with that exception.

**emits** the future completes

**completes** after the future has completed

## unfold

Stream the result of a function as long as it returns a `Some`, the value inside the option consists of a tuple where the first value is a state passed back into the next call to the function allowing to pass a state. The first invocation of the provided fold function will receive the `zero` state.

Can be used to implement many stateful sources without having to touch the more low level `GraphStage` API.

**emits** when there is demand and the unfold function over the previous state returns non empty value

**completes** when the unfold function returns an empty value

## unfoldAsync

Just like `unfold` but the fold function returns a `Future` which will cause the source to complete or emit when it completes.

Can be used to implement many stateful sources without having to touch the more low level `GraphStage` API.

**emits** when there is demand and unfold state returned future completes with some value

**completes** when the future returned by the unfold function completes with an empty value

## empty

Complete right away without ever emitting any elements. Useful when you have to provide a source to an API but there are no elements to emit.

**emits** never

**completes** directly

## maybe

Materialize a `Promise[Option[T]]` that if completed with a `Some[T]` will emit that T and then complete the stream, or if completed with `None` complete the stream right away.

**emits** when the returned promise is completed with some value

**completes** after emitting some value, or directly if the promise is completed with no value

## failed

Fail directly with a user specified exception.

**emits** never

**completes** fails the stream directly with the given exception

## actorPublisher

Wrap an actor extending `ActorPublisher` as a source.

**emits** depends on the actor implementation

**completes** when the actor stops

## actorRef

Materialize an `ActorRef`, sending messages to it will emit them on the stream. The actor contain a buffer but since communication is one way, there is no back pressure. Handling overflow is done by either dropping elements or failing the stream, the strategy is chosen by the user.

**emits** when there is demand and there are messages in the buffer or a message is sent to the actorref

**completes** when the actorref is sent `akka.actor.Status.Success` or `PoisonPill`

## combine

Combine several sources, using a given strategy such as merge or concat, into one source.

**emits** when there is demand, but depending on the strategy

**completes** when all sources has completed

## unfoldResource

Wrap any resource that can be opened, queried for next element (in a blocking way) and closed using three distinct functions into a source.

**emits** when there is demand and read function returns value

**completes** when read function returns `None`

## unfoldAsyncResource

Wrap any resource that can be opened, queried for next element (in a blocking way) and closed using three distinct functions into a source. Functions return `Future` to achieve asynchronous processing

**emits** when there is demand and `Future` from read function returns value

**completes** when `Future` from read function returns `None`

## queue

Materialize a `SourceQueue` onto which elements can be pushed for emitting from the source. The queue contains a buffer, if elements are pushed onto the queue faster than the source is consumed the overflow will be handled with a strategy specified by the user. Functionality for tracking when an element has been emitted is available through `SourceQueue.offer`.

**emits** when there is demand and the queue contains elements

**completes** when downstream completes

## asSubscriber

Integration with Reactive Streams, materializes into a `org.reactivestreams.Subscriber`.

## fromPublisher

Integration with Reactive Streams, subscribes to a `org.reactivestreams.Publisher`.

## zipN

Combine the elements of multiple streams into a stream of sequences.

**emits** when all of the inputs has an element available

**completes** when any upstream completes

## zipWithN

Combine the elements of multiple streams into a stream of sequences using a combiner function.

**emits** when all of the inputs has an element available

**completes** when any upstream completes

# Sink stages

These built-in sinks are available from `akka.stream.scaladsl.Sink`:

## head

Materializes into a `Future` which completes with the first value arriving, after this the stream is canceled. If no element is emitted, the future is be failed.

**cancels** after receiving one element

**backpressures** never

## headOption

Materializes into a `Future[Option[T]]` which completes with the first value arriving wrapped in a `Some`, or a `None` if the stream completes without any elements emitted.

**cancels** after receiving one element

**backpressures** never

## last

Materializes into a `Future` which will complete with the last value emitted when the stream completes. If the stream completes with no elements the future is failed.

**cancels** never

**backpressures** never

## lastOption

Materialize a `Future[Option[T]]` which completes with the last value emitted wrapped in an `Some` when the stream completes. if the stream completes with no elements the future is completed with `None`.

**cancels** never

**backpressures** never

## ignore

Consume all elements but discards them. Useful when a stream has to be consumed but there is no use to actually do anything with the elements.

**cancels** never

**backpressures** never

## cancelled

Immediately cancel the stream

**cancels** immediately

## seq

Collect values emitted from the stream into a collection, the collection is available through a `Future` or which completes when the stream completes. Note that the collection is bounded to `Int.MaxValue`, if more element are emitted the sink will cancel the stream

**cancels** If too many values are collected

## foreach

Invoke a given procedure for each element received. Note that it is not safe to mutate shared state from the procedure.

The sink materializes into a `Future[Option[Done]]` which completes when the stream completes, or fails if the stream fails.

Note that it is not safe to mutate state from the procedure.

**cancels** never

**backpressures** when the previous procedure invocation has not yet completed

## foreachParallel

Like `foreach` but allows up to `parallellism` procedure calls to happen in parallel.

**cancels** never

**backpressures** when the previous parallel procedure invocations has not yet completed

## onComplete

Invoke a callback when the stream has completed or failed.

**cancels** never

**backpressures** never

## fold

Fold over emitted element with a function, where each invocation will get the new element and the result from the previous fold invocation. The first invocation will be provided the `zero` value.

Materializes into a future that will complete with the last state when the stream has completed.

This stage allows combining values into a result without a global mutable state by instead passing the state along between invocations.

**cancels** never

**backpressures** when the previous fold function invocation has not yet completed

## reduce

Apply a reduction function on the incoming elements and pass the result to the next invocation. The first invocation receives the two first elements of the flow.

Materializes into a future that will be completed by the last result of the reduction function.

**cancels** never

**backpressures** when the previous reduction function invocation has not yet completed

## combine

Combine several sinks into one using a user specified strategy

**cancels** depends on the strategy

**backpressures** depends on the strategy

### actorRef

Send the elements from the stream to an `ActorRef`. No backpressure so care must be taken to not overflow the inbox.

**cancels** when the actor terminates

**backpressures** never

### actorRefWithAck

Send the elements from the stream to an `ActorRef` which must then acknowledge reception after completing a message, to provide back pressure onto the sink.

**cancels** when the actor terminates

**backpressures** when the actor acknowledgement has not arrived

### actorSubscriber

Create an actor from a `Props` upon materialization, where the actor implements `ActorSubscriber`, which will receive the elements from the stream.

Materializes into an `ActorRef` to the created actor.

**cancels** when the actor terminates

**backpressures** depends on the actor implementation

### asPublisher

Integration with Reactive Streams, materializes into a `org.reactivestreams.Publisher`.

### fromSubscriber

Integration with Reactive Streams, wraps a `org.reactivestreams.Subscriber` as a sink

## Additional Sink and Source converters

Sources and sinks for integrating with `java.io.InputStream` and `java.io.OutputStream` can be found on `StreamConverters`. As they are blocking APIs the implementations of these stages are run on a separate dispatcher configured through the `akka.stream.blocking-io-dispatcher`.

## fromOutputStream

Create a sink that wraps an `OutputStream`. Takes a function that produces an `OutputStream`, when the sink is materialized the function will be called and bytes sent to the sink will be written to the returned `OutputStream`.

Materializes into a `Future` which will complete with a `IOResult` when the stream completes.

Note that a flow can be materialized multiple times, so the function producing the `OutputStream` must be able to handle multiple invocations.

The `OutputStream` will be closed when the stream that flows into the `Sink` is completed, and the `Sink` will cancel its inflow when the `OutputStream` is no longer writable.

## asInputStream

Create a sink which materializes into an `InputStream` that can be read to trigger demand through the sink. Bytes emitted through the stream will be available for reading through the `InputStream`

The `InputStream` will be ended when the stream flowing into this `Sink` completes, and the closing the `InputStream` will cancel the inflow of this `Sink`.

## fromInputStream

Create a source that wraps an `InputStream`. Takes a function that produces an `InputStream`, when the source is materialized the function will be called and bytes from the `InputStream` will be emitted into the stream.

Materializes into a `Future` which will complete with a `IOResult` when the stream completes.

Note that a flow can be materialized multiple times, so the function producing the `InputStream` must be able to handle multiple invocations.

The `InputStream` will be closed when the `Source` is canceled from its downstream, and reaching the end of the `InputStream` will complete the `Source`.

## asOutputStream

Create a source that materializes into an `OutputStream`. When bytes are written to the `OutputStream` they are emitted from the source.

The `OutputStream` will no longer be writable when the `Source` has been canceled from its downstream, and closing the `OutputStream` will complete the `Source`.

### asJavaStream

Create a sink which materializes into Java 8 `Stream` that can be run to trigger demand through the sink. Elements emitted through the stream will be available for reading through the Java 8 `Stream`.

The Java 8 `Stream` will be ended when the stream flowing into this `Sink` completes, and closing the Java `Stream` will cancel the inflow of this `Sink`. Java `Stream` throws exception in case reactive stream failed.

Be aware that Java `Stream` blocks current thread while waiting on next element from downstream.

### fromJavaStream

Create a source that wraps a Java 8 `Stream`. `Source` uses a stream iterator to get all its elements and send them downstream on demand.

### javaCollector

Create a sink which materializes into a `Future` which will be completed with a result of the Java 8 `Collector` transformation and reduction operations. This allows usage of Java 8 streams transformations for reactive streams. The `Collector` will trigger demand downstream. Elements emitted through the stream will be accumulated into a mutable result container, optionally transformed into a final representation after all input elements have been processed. The `Collector` can also do reduction at the end. Reduction processing is performed sequentially

Note that a flow can be materialized multiple times, so the function producing the `Collector` must be able to handle multiple invocations.

### javaCollectorParallelUnordered

Create a sink which materializes into a `Future` which will be completed with a result of the Java 8 `Collector` transformation and reduction operations. This allows usage of Java 8 streams transformations for reactive streams. The `Collector` is triggering demand downstream. Elements emitted through the stream will be accumulated into a mutable result container, optionally transformed into a final representation after all input elements have been processed. The `Collector` can also do reduction at the end. Reduction processing is performed in parallel based on graph `Balance`.

Note that a flow can be materialized multiple times, so the function producing the `Collector` must be able to handle multiple invocations.

## File IO Sinks and Sources

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

### fromFile

Emit the contents of a file, as `ByteString` s, materializes into a `Future` which will be completed with a `IOResult` upon reaching the end of the file or if there is a failure.

### toFile

Create a sink which will write incoming `ByteString` s to a given file.

## Flow stages

All flows by default backpressure if the computation they encapsulate is not fast enough to keep up with the rate of incoming elements from the preceding stage. There are differences though how the different stages handle when some of their downstream stages backpressure them.

Most stages stop and propagate the failure downstream as soon as any of their upstreams emit a failure. This happens to ensure reliable teardown of streams and cleanup when failures happen. Failures are meant to be to model unrecoverable conditions, therefore they are always eagerly propagated. For in-band error handling of normal errors (dropping elements if a map fails for example) you should use the supervision support, or explicitly wrap your element types in a proper container that can express error or success states (for example `Try` in Scala).

## Simple processing stages

These stages can transform the rate of incoming elements since there are stages that emit multiple elements for a single input (e.g. mapConcat') or consume multiple elements before emitting one output (e.g. ``filter`). However, these rate transformations are data-driven, i.e. it is the incoming elements that define how the rate is affected. This is in contrast with Backpressure aware stages which can change their processing behavior depending on being backpressured by downstream or not.

### map

Transform each element in the stream by calling a mapping function with it and passing the returned value downstream.

**emits** when the mapping function returns an element

**backpressures** when downstream backpressures

**completes** when upstream completes

### mapConcat

Transform each element into zero or more elements that are individually passed downstream.

**emits** when the mapping function returns an element or there are still remaining elements from the previously calculated collection

**backpressures** when downstream backpressures or there are still available elements from the previously calculated collection

**completes** when upstream completes and all remaining elements has been emitted

## statefulMapConcat

Transform each element into zero or more elements that are individually passed downstream. The difference to `mapConcat` is that the transformation function is created from a factory for every materialization of the flow.

**emits** when the mapping function returns an element or there are still remaining elements from the previously calculated collection

**backpressures** when downstream backpressures or there are still available elements from the previously calculated collection

**completes** when upstream completes and all remaining elements has been emitted

## filter

Filter the incoming elements using a predicate. If the predicate returns true the element is passed downstream, if it returns false the element is discarded.

**emits** when the given predicate returns true for the element

**backpressures** when the given predicate returns true for the element and downstream backpressures

**completes** when upstream completes

## collect

Apply a partial function to each incoming element, if the partial function is defined for a value the returned value is passed downstream. Can often replace `filter` followed by `map` to achieve the same in one single stage.

**emits** when the provided partial function is defined for the element

**backpressures** the partial function is defined for the element and downstream backpressures

**completes** when upstream completes

## grouped

Accumulate incoming events until the specified number of elements have been accumulated and then pass the collection of elements downstream.

**emits** when the specified number of elements has been accumulated or upstream completed

**backpressures** when a group has been assembled and downstream backpressures

**completes** when upstream completes

## sliding

Provide a sliding window over the incoming stream and pass the windows as groups of elements downstream.

Note: the last window might be smaller than the requested size due to end of stream.

**emits** the specified number of elements has been accumulated or upstream completed

**backpressures** when a group has been assembled and downstream backpressures

**completes** when upstream completes

## scan

Emit its current value which starts at `zero` and then applies the current and next value to the given function emitting the next current value.

Note that this means that scan emits one element downstream before and upstream elements will not be requested until the second element is required from downstream.

**emits** when the function scanning the element returns a new element

**backpressures** when downstream backpressures

**completes** when upstream completes

## fold

Start with current value `zero` and then apply the current and next value to the given function, when upstream complete the current value is emitted downstream.

**emits** when upstream completes

**backpressures** when downstream backpressures

**completes** when upstream completes

## drop

Drop `n` elements and then pass any subsequent element downstream.

**emits** when the specified number of elements has been dropped already

**backpressures** when the specified number of elements has been dropped and downstream backpressures

**completes** when upstream completes

## take

Pass `n` incoming elements downstream and then complete

**emits** while the specified number of elements to take has not yet been reached

**backpressures** when downstream backpressures

**completes** when the defined number of elements has been taken or upstream completes

## 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.

**emits** while the predicate is true and until the first false result

**backpressures** when downstream backpressures

**completes** when predicate returned false or upstream completes

## dropWhile

Drop elements as long as a predicate function return true for the element

**emits** when the predicate returned false and for all following stream elements

**backpressures** predicate returned false and downstream backpressures

**completes** when upstream completes

## recover

Allow sending of one last element downstream when a failure has happened upstream.

**emits** when the element is available from the upstream or upstream is failed and pf returns an element

**backpressures** when downstream backpressures, not when failure happened

**completes** when upstream completes or upstream failed with exception pf can handle

## recoverWith

Allow switching to alternative Source when a failure has happened upstream.

**emits** the element is available from the upstream or upstream is failed and pf returns alternative Source

**backpressures** downstream backpressures, after failure happened it backprssures to alternative Source

**completes** upstream completes or upstream failed with exception pf can handle

## detach

Detach upstream demand from downstream demand without detaching the stream rates.

**emits** when the upstream stage has emitted and there is demand

**backpressures** when downstream backpressures

**completes** when upstream completes

## throttle

Limit the throughput to a specific number of elements per time unit, or a specific total cost per time unit, where a function has to be provided to calculate the individual cost of each element.

**emits** when upstream emits an element and configured time per each element elapsed

**backpressures** when downstream backpressures

**completes** when upstream completes

# Asynchronous processing stages

These stages encapsulate an asynchronous computation, properly handling backpressure while taking care of the asynchronous operation at the same time (usually handling the completion of a Future).

## mapAsync

Pass incoming elements to a function that return a `Future` result. When the future arrives the result is passed downstream. Up to `n` elements can be processed concurrently, but regardless of their completion time the incoming order will be kept when results complete. For use cases where order does not mather `mapAsyncUnordered` can be used.

If a Future fails, the stream also fails (unless a different supervision strategy is applied)

**emits** when the Future returned by the provided function finishes for the next element in sequence

**backpressures** when the number of futures reaches the configured parallelism and the downstream backpressures

**completes** when upstream completes and all futures has been completed and all elements has been emitted

## mapAsyncUnordered

Like `mapAsync` but `Future` results are passed downstream as they arrive regardless of the order of the elements that triggered them.

If a Future fails, the stream also fails (unless a different supervision strategy is applied)

**emits** any of the Futures returned by the provided function complete

**backpressures** when the number of futures reaches the configured parallelism and the downstream backpressures

**completes** upstream completes and all futures has been completed and all elements has been emitted

# Timer driven stages

These stages process elements using timers, delaying, dropping or grouping elements for certain time durations.

### takeWithin

Pass elements downstream within a timeout and then complete.

**emits** when an upstream element arrives

**backpressures** downstream backpressures

**completes** upstream completes or timer fires

### dropWithin

Drop elements until a timeout has fired

**emits** after the timer fired and a new upstream element arrives

**backpressures** when downstream backpressures

**completes** upstream completes

### groupedWithin

Chunk up the stream into groups of elements received within a time window, or limited by the given number of elements, whichever happens first.

**emits** when the configured time elapses since the last group has been emitted

**backpressures** when the group has been assembled (the duration elapsed) and downstream backpressures

**completes** when upstream completes

### initialDelay

Delay the initial element by a user specified duration from stream materialization.

**emits** upstream emits an element if the initial delay already elapsed

**backpressures** downstream backpressures or initial delay not yet elapsed

**completes** when upstream completes

### delay

Delay every element passed through with a specific duration.

**emits** there is a pending element in the buffer and configured time for this element elapsed

**backpressures** differs, depends on `OverflowStrategy` set

**completes** when upstream completes and buffered elements has been drained

## Backpressure aware stages

These stages are aware of the backpressure provided by their downstreams and able to adapt their behavior to that signal.

### conflate

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure. The summary value must be of the same type as the incoming elements, for example the sum or average of incoming numbers, if aggregation should lead to a different type `conflateWithSeed` can be used:

**emits** when downstream stops backpressuring and there is a conflated element available

**backpressures** when the aggregate function cannot keep up with incoming elements

**completes** when upstream completes

### conflateWithSeed

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure. When backpressure starts or there is no backpressure element is passed into a `seed` function to transform it to the summary type.

**emits** when downstream stops backpressuring and there is a conflated element available

**backpressures** when the aggregate or seed functions cannot keep up with incoming elements

**completes** when upstream completes

### batch

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure and a maximum number of batched elements is not yet reached. When the maximum number is reached and downstream still backpressures batch will also backpressure.

When backpressure starts or there is no backpressure element is passed into a `seed` function to transform it to the summary type.

Will eagerly pull elements, this behavior may result in a single pending (i.e. buffered) element which cannot be aggregated to the batched value.

**emits** when downstream stops backpressuring and there is a batched element available

**backpressures** when batched elements reached the max limit of allowed batched elements & downstream backpressures

**completes** when upstream completes and a "possibly pending" element was drained

## batchWeighted

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure and a maximum weight batched elements is not yet reached. The weight of each element is determined by applying `costFn`. When the maximum total weight is reached and downstream still backpressures batch will also backpressure.

Will eagerly pull elements, this behavior may result in a single pending (i.e. buffered) element which cannot be aggregated to the batched value.

**emits** downstream stops backpressuring and there is a batched element available

**backpressures** batched elements reached the max weight limit of allowed batched elements & downstream backpressures

**completes** upstream completes and a "possibly pending" element was drained

## expand

Allow for a faster downstream by expanding the last incoming element to an `Iterator`. For example `Iterator.continually(element)` to keep repating the last incoming element.

**emits** when downstream stops backpressuring

**backpressures** when downstream backpressures

**completes** when upstream completes

## buffer (Backpressure)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full backpressure is applied.

**emits** when downstream stops backpressuring and there is a pending element in the buffer

**backpressures** when buffer is full

**completes** when upstream completes and buffered elements has been drained

### buffer (Drop)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full elements are dropped according to the specified `OverflowStrategy`:

- `dropHead` drops the oldest element in the buffer to make space for the new element
- `dropTail` drops the youngest element in the buffer to make space for the new element
- `dropBuffer` drops the entire buffer and buffers the new element
- `dropNew` drops the new element

**emits** when downstream stops backpressuring and there is a pending element in the buffer

**backpressures** never (when dropping cannot keep up with incoming elements)

**completes** upstream completes and buffered elements has been drained

### buffer (Fail)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full the stage fails the flow with a `BufferOverflowException`.

**emits** when downstream stops backpressuring and there is a pending element in the buffer

**backpressures** never, fails the stream instead of backpressuring when buffer is full

**completes** when upstream completes and buffered elements has been drained

## Nesting and flattening stages

These stages either take a stream and turn it into a stream of streams (nesting) or they take a stream that contains nested streams and turn them into a stream of elements instead (flattening).

### prefixAndTail

Take up to n elements from the stream (less than n only if the upstream completes before emitting n elements) and returns a pair containing a strict sequence of the taken element and a stream representing the remaining elements.

**emits** when the configured number of prefix elements are available. Emits this prefix, and the rest as a substream

**backpressures** when downstream backpressures or substream backpressures

**completes** when prefix elements has been consumed and substream has been consumed

## groupBy

Demultiplex the incoming stream into separate output streams.

**emits** an element for which the grouping function returns a group that has not yet been created. Emits the new group there is an element pending for a group whose substream backpressures

**completes** when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

## splitWhen

Split off elements into a new substream whenever a predicate function return `true`.

**emits** an element for which the provided predicate is true, opening and emitting a new substream for subsequent elements

**backpressures** when there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures

**completes** when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

## splitAfter

End the current substream whenever a predicate returns `true`, starting a new substream for the next element.

**emits** when an element passes through. When the provided predicate is true it emits the element * and opens a new substream for subsequent element

**backpressures** when there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures

**completes** when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

## flatMapConcat

Transform each input element into a `Source` whose elements are then flattened into the output stream through concatenation. This means each source is fully consumed before consumption of the next source starts.

**emits** when the current consumed substream has an element available

**backpressures** when downstream backpressures

**completes** when upstream completes and all consumed substreams complete

### flatMapMerge

Transform each input element into a `Source` whose elements are then flattened into the output stream through merging. The maximum number of merged sources has to be specified.

**emits** when one of the currently consumed substreams has an element available

**backpressures** when downstream backpressures

**completes** when upstream completes and all consumed substreams complete

## Time aware stages

Those stages operate taking time into consideration.

### initialTimeout

If the first element has not passed through this stage before the provided timeout, the stream is failed with a `TimeoutException`.

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses before first element arrives

**cancels** when downstream cancels

### completionTimeout

If the completion of the stream does not happen until the provided timeout, the stream is failed with a `TimeoutException`.

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses before upstream completes

**cancels** when downstream cancels

## idleTimeout

If the time between two processed elements exceeds the provided timeout, the stream is failed with a `TimeoutException`. The timeout is checked periodically, so the resolution of the check is one period (equals to timeout value).

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses between two emitted elements

**cancels** when downstream cancels

## backpressureTimeout

If the time between the emission of an element and the following downstream demand exceeds the provided timeout, the stream is failed with a `TimeoutException`. The timeout is checked periodically, so the resolution of the check is one period (equals to timeout value).

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses between element emission and downstream demand.

**cancels** when downstream cancels

## keepAlive

Injects additional (configured) elements if upstream does not emit for a configured amount of time.

**emits** when upstream emits an element or if the upstream was idle for the configured period

**backpressures** when downstream backpressures

**completes** when upstream completes

**cancels** when downstream cancels

### initialDelay

Delays the initial element by the specified duration.

**emits** when upstream emits an element if the initial delay is already elapsed

**backpressures** when downstream backpressures or initial delay is not yet elapsed

**completes** when upstream completes

**cancels** when downstream cancels

## Fan-in stages

These stages take multiple streams as their input and provide a single output combining the elements from all of the inputs in different ways.

### merge

Merge multiple sources. Picks elements randomly if all sources has elements ready.

**emits** when one of the inputs has an element available

**backpressures** when downstream backpressures

**completes** when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

### mergeSorted

Merge multiple sources. Waits for one element to be ready from each input stream and emits the smallest element.

**emits** when all of the inputs have an element available

**backpressures** when downstream backpressures

**completes** when all upstreams complete

### mergePreferred

Merge multiple sources. Prefer one source if all sources has elements ready.

**emits** when one of the inputs has an element available, preferring a defined input if multiple have elements available

**backpressures** when downstream backpressures

**completes** when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

### zip

Combines elements from each of multiple sources into tuples and passes the tuples downstream.

**emits** when all of the inputs have an element available

**backpressures** when downstream backpressures

**completes** when any upstream completes

### zipWith

Combines elements from multiple sources through a `combine` function and passes the returned value downstream.

**emits** when all of the inputs have an element available

**backpressures** when downstream backpressures

**completes** when any upstream completes

### concat

After completion of the original upstream the elements of the given source will be emitted.

**emits** when the current stream has an element available; if the current input completes, it tries the next one

**backpressures** when downstream backpressures

**completes** when all upstreams complete

### prepend

Prepends the given source to the flow, consuming it until completion before the original source is consumed.

If materialized values needs to be collected `prependMat` is available.

**emits** when the given stream has an element available; if the given input completes, it tries the current one

**backpressures** when downstream backpressures

**completes** when all upstreams complete

### interleave

Emits a specifiable number of elements from the original source, then from the provided source and repeats. If one source completes the rest of the other stream will be emitted.

**emits** when element is available from the currently consumed upstream

**backpressures** when upstream backpressures

**completes** when both upstreams have completed

## Fan-out stages

These have one input and multiple outputs. They might route the elements between different outputs, or emit elements on multiple outputs at the same time.

### unzip

Takes a stream of two element tuples and unzips the two elements ino two different downstreams.

**emits** when all of the outputs stops backpressuring and there is an input element available

**backpressures** when any of the outputs backpressures

**completes** when upstream completes

### unzipWith

Splits each element of input into multiple downstreams using a function

**emits** when all of the outputs stops backpressuring and there is an input element available

**backpressures** when any of the outputs backpressures

**completes** when upstream completes

### broadcast

Emit each incoming element each of `n` outputs.

**emits** when all of the outputs stops backpressuring and there is an input element available

**backpressures** when any of the outputs backpressures

**completes** when upstream completes

### balance

Fan-out the stream to several streams. Each upstream element is emitted to the first available downstream consumer.

**emits** when any of the outputs stops backpressuring; emits the element to the first available output

**backpressures** when all of the outputs backpressure

**completes** when upstream completes

### partition

Fan-out the stream to several streams. Each upstream element is emitted to one downstream consumer according to the partitioner function applied to the element.

**emits** when the chosen output stops backpressuring and there is an input element available

**backpressures** when the chosen output backpressures

**completes** when upstream completes and no output is pending

## 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. The stage otherwise passes through elements unchanged.

**emits** when input has an element available

**backpressures** when output backpressures

**completes** when upstream completes

### 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.

**emits** when upstream emits an element

**backpressures** when downstream **backpressures**

**completes** when upstream completes

# Streams Cookbook

## Introduction

This is a collection of patterns to demonstrate various usage of the Akka Streams API by solving small targeted problems in the format of "recipes". The purpose of this page is to give inspiration and ideas how to approach various small tasks involving streams. The recipes in this page can be used directly as-is, but they are most powerful as starting points: customization of the code snippets is warmly encouraged.

This part also serves as supplementary material for the main body of documentation. It is a good idea to have this page open while reading the manual and look for examples demonstrating various streaming concepts as they appear in the main body of documentation.

If you need a quick reference of the available processing stages used in the recipes see Overview of built-in stages and their semantics.

## Working with Flows

In this collection we show simple recipes that involve linear flows. The recipes in this section are rather general, more targeted recipes are available as separate sections (Buffers and working with rate, Working with streaming IO).

## Logging elements of a stream

**Situation:** During development it is sometimes helpful to see what happens in a particular section of a stream.

The simplest solution is to simply use a `map` operation and use `println` to print the elements received to the console. While this recipe is rather simplistic, it is often suitable for a quick debug session.

```
1. val loggedSource = mySource.map { elem => println(elem); elem }
```

Another approach to logging is to use `log()` operation which allows configuring logging for elements flowing through the stream as well as completion and erroring.

```
1. // customise log levels
2. mySource.log("before-map")
3.   .withAttributes(Attributes.logLevels(onElement = Logging.WarningLevel))
4.   .map(analyse)
5.
6. // or provide custom logging adapter
7. implicit val adapter = Logging(system, "customLogger")
8. mySource.log("custom")
```

## Flattening a stream of sequences

**Situation:** A stream is given as a stream of sequence of elements, but a stream of elements needed instead, streaming all the nested elements inside the sequences separately.

The `mapConcat` operation can be used to implement a one-to-many transformation of elements using a mapper function in the form of `In => immutable.Seq[Out]`. In this case we want to map a `Seq` of elements to the elements in the collection itself, so we can just call `mapConcat(identity)`.

```
1. val myData: Source[List[Message], NotUsed] = someDataSource
2. val flattened: Source[Message, NotUsed] = myData.mapConcat(identity)
```

## Draining a stream to a strict collection

**Situation:** A possibly unbounded sequence of elements is given as a stream, which needs to be collected into a Scala collection while ensuring boundedness

A common situation when working with streams is one where we need to collect incoming elements into a Scala collection. This operation is supported via `Sink.seq` which materializes into a `Future[Seq[T]]`.

The function `limit` or `take` should always be used in conjunction in order to guarantee stream boundedness, thus preventing the program from running out of memory.

For example, this is best avoided:

```
1. // Dangerous: might produce a collection with 2 billion elements!
2. val f: Future[Seq[String]] = mySource.runWith(Sink.seq)
```

Rather, use `limit` or `take` to ensure that the resulting `Seq` will contain only up to `max` elements:

```
1. val MAX_ALLOWED_SIZE = 100
2.
3. // OK. Future will fail with a `StreamLimitReachedException`
4. // if the number of incoming elements is larger than max
5. val limited: Future[Seq[String]] =
6.   mySource.limit(MAX_ALLOWED_SIZE).runWith(Sink.seq)
7.
8. // OK. Collect up until max-th elements only, then cancel upstream
9. val ignoreOverflow: Future[Seq[String]] =
10.   mySource.take(MAX_ALLOWED_SIZE).runWith(Sink.seq)
```

## Calculating the digest of a ByteString stream

**Situation:** A stream of bytes is given as a stream of `ByteStrings` and we want to calculate the cryptographic digest of the stream.

This recipe uses a `GraphStage` to host a mutable `MessageDigest` class (part of the Java Cryptography API) and update it with the bytes arriving from the stream. When the stream starts, the `onPull` handler of the stage is called, which just bubbles up the `pull` event to its upstream. As a response to this pull, a ByteString

chunk will arrive (`onPush`) which we use to update the digest, then it will pull for the next chunk.

Eventually the stream of `ByteStrings` depletes and we get a notification about this event via `onUpstreamFinish`. At this point we want to emit the digest value, but we cannot do it with `push` in this handler directly since there may be no downstream demand. Instead we call `emit` which will temporarily replace the handlers, emit the provided value when demand comes in and then reset the stage state. It will then complete the stage.

```
1. import akka.stream.stage._
2. class DigestCalculator(algorithm: String) extends GraphStage[FlowShape[ByteString, ByteString]] {
3.   val in = Inlet[ByteString]("DigestCalculator.in")
4.   val out = Outlet[ByteString]("DigestCalculator.out")
5.   override val shape = FlowShape.of(in, out)
6.
7.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {
8.     val digest = MessageDigest.getInstance(algorithm)
9.
10.    setHandler(out, new OutHandler {
11.      override def onPull(): Trigger = {
12.        pull(in)
13.      }
14.    })
15.
16.    setHandler(in, new InHandler {
17.      override def onPush(): Trigger = {
18.        val chunk = grab(in)
19.        digest.update(chunk.toArray)
20.        pull(in)
21.      }
22.
23.      override def onUpstreamFinish(): Unit = {
24.        emit(out, ByteString(digest.digest()))
25.        completeStage()
26.      }
27.    })
28.
```

```
  }
}
val digest: Source[ByteString, NotUsed] = data.via(new DigestCalculator("SHA-256"))
```

## Parsing lines from a stream of ByteStrings

**Situation:** A stream of bytes is given as a stream of `ByteStrings` containing lines terminated by line ending characters (or, alternatively, containing binary frames delimited by a special delimiter byte sequence) which needs to be parsed.

The `Framing` helper object contains a convenience method to parse messages from a stream of `ByteStrings`:

```
import akka.stream.scaladsl.Framing
val linesStream = rawData.via(Framing.delimiter(
  ByteString("\r\n"), maximumFrameLength = 100, allowTruncation = true))
  .map(_.utf8String)
```

## Implementing reduce-by-key

**Situation:** Given a stream of elements, we want to calculate some aggregated value on different subgroups of the elements.

The "hello world" of reduce-by-key style operations is *wordcount* which we demonstrate below. Given a stream of words we first create a new stream that groups the words according to the `identity` function, i.e. now we have a stream of streams, where every substream will serve identical words.

To count the words, we need to process the stream of streams (the actual groups containing identical words). `groupBy` returns a `SubFlow`, which means that we transform the resulting substreams directly. In this case we use the `reduce` combinator to aggregate the word itself and the number of its occurrences within a tuple `(String, Integer)`. Each substream will then emit one final value—precisely such a pair—when the overall input completes. As a last step we merge back these values from the substreams into one single output stream.

One noteworthy detail pertains to the `MaximumDistinctWords` parameter: this defines the breadth of the groupBy and merge operations. Akka Streams is focused on bounded resource consumption and the number of concurrently open inputs to the merge operator describes the amount of resources needed by the merge itself. Therefore only a finite number of substreams can be active at any given time. If the `groupBy` operator encounters more keys than this number then the stream cannot continue without violating its resource bound, in this case `groupBy` will terminate with a failure.

```
val counts: Source[(String, Int), NotUsed] = words
  // split the words into separate streams first
```

```
  .groupBy(MaximumDistinctWords, identity)
  //transform each element to pair with number of words in it
  .map(_ -> 1)
  // add counting logic to the streams
  .reduce((l, r) => (l._1, l._2 + r._2))
  // get a stream of word counts
  .mergeSubstreams
```

By extracting the parts specific to *wordcount* into

- a `groupKey` function that defines the groups
- a `map` map each element to value that is used by the reduce on the substream
- a `reduce` function that does the actual reduction

we get a generalized version below:

```
def reduceByKey[In, K, Out](
  maximumGroupSize: Int,
  groupKey:         (In) => K,
  map:              (In) => Out)(reduce: (Out, Out) => Out): Flow[In, (K, Out), NotUsed] = {

  Flow[In]
    .groupBy[K](maximumGroupSize, groupKey)
    .map(e => groupKey(e) -> map(e))
    .reduce((l, r) => l._1 -> reduce(l._2, r._2))
    .mergeSubstreams
}

val wordCounts = words.via(
  reduceByKey(
    MaximumDistinctWords,
    groupKey = (word: String) => word,
    map = (word: String) => 1)((left: Int, right: Int) => left + right))
```

**Note**

Please note that the reduce-by-key version we discussed above is sequential in reading the overall input stream, in other words it is **NOT** a parallelization pattern like MapReduce and similar frameworks.

### Sorting elements to multiple groups with groupBy

**Situation:** The `groupBy` operation strictly partitions incoming elements, each element belongs to exactly one group. Sometimes we want to map elements into multiple groups simultaneously.

To achieve the desired result, we attack the problem in two steps:

- first, using a function `topicMapper` that gives a list of topics (groups) a message belongs to, we transform our stream of `Message` to a stream of `(Message, Topic)` where for each topic the message belongs to a separate pair will be emitted. This is achieved by using `mapConcat`
- Then we take this new stream of message topic pairs (containing a separate pair for each topic a given message belongs to) and feed it into groupBy, using the topic as the group key.

```
1. val topicMapper: (Message) => immutable.Seq[Topic] = extractTopics
2.
3. val messageAndTopic: Source[(Message, Topic), NotUsed] = elems.mapConcat { msg: Message =>
4.   val topicsForMessage = topicMapper(msg)
5.   // Create a (Msg, Topic) pair for each of the topics
6.   // the message belongs to
7.   topicsForMessage.map(msg -> _)
8. }
9.
10. val multiGroups = messageAndTopic
11.   .groupBy(2, _._2).map {
12.     case (msg, topic) =>
13.       // do what needs to be done
14.   }
```

# Working with Graphs

In this collection we show recipes that use stream graph elements to achieve various goals.

## Triggering the flow of elements programmatically

**Situation:** Given a stream of elements we want to control the emission of those elements according to a trigger signal. In other words, even if the stream would be able to flow (not being backpressured) we want to hold back elements until a trigger signal arrives.

This recipe solves the problem by simply zipping the stream of `Message` elements with the stream of `Trigger` signals. Since `Zip` produces pairs, we simply map the output stream selecting the first element of the pair.

```
1. val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
2.   import GraphDSL.Implicits._
3.   val zip = builder.add(Zip[Message, Trigger]())
4.   elements ~> zip.in0
5.   triggerSource ~> zip.in1
6.   zip.out ~> Flow[(Message, Trigger)].map { case (msg, trigger) => msg } ~> sink
7.   ClosedShape
8. })
```

Alternatively, instead of using a `Zip`, and then using `map` to get the first element of the pairs, we can avoid creating the pairs in the first place by using `ZipWith` which takes a two argument function to produce the output element. If this function would return a pair of the two argument it would be exactly the behavior of `Zip` so `ZipWith` is a generalization of zipping.

```
1. val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
2.   import GraphDSL.Implicits._
3.   val zip = builder.add(ZipWith((msg: Message, trigger: Trigger) => msg))
4.
5.   elements ~> zip.in0
6.   triggerSource ~> zip.in1
7.   zip.out ~> sink
8.   ClosedShape
9. })
```

## Balancing jobs to a fixed pool of workers

**Situation:** Given a stream of jobs and a worker process expressed as a `Flow` create a pool of workers that automatically balances incoming jobs to available workers, then merges the results.

We will express our solution as a function that takes a worker flow and the number of workers to be allocated and gives a flow that internally contains a pool of these workers. To achieve the desired result we will create a `Flow` from a graph.

The graph consists of a `Balance` node which is a special fan-out operation that tries to route elements to available downstream consumers. In a `for` loop we wire all of our desired workers as outputs of this balancer element, then we wire the outputs of these workers to a `Merge` element that will collect the results from the workers.

To make the worker stages run in parallel we mark them as asynchronous with async.

```
1. def balancer[In, Out](worker: Flow[In, Out, Any], workerCount: Int): Flow[In, Out, NotUsed] = {
2.   import GraphDSL.Implicits._
3.
4.   Flow.fromGraph(GraphDSL.create() { implicit b =>
5.     val balancer = b.add(Balance[In](workerCount, waitForAllDownstreams = true))
6.     val merge = b.add(Merge[Out](workerCount))
7.
8.     for (_ <- 1 to workerCount) {
9.       // for each worker, add an edge from the balancer to the worker, then wire
10.      // it to the merge element
11.      balancer ~> worker.async ~> merge
12.    }
13.
14.    FlowShape(balancer.in, merge.out)
15.  })
16. }
17.
18. val processedJobs: Source[Result, NotUsed] = myJobs.via(balancer(worker, 3))
```

# Working with rate

This collection of recipes demonstrate various patterns where rate differences between upstream and downstream needs to be handled by other strategies than simple backpressure.

## Dropping elements

**Situation:** Given a fast producer and a slow consumer, we want to drop elements if necessary to not slow down the producer too much.

This can be solved by using a versatile rate-transforming operation, `conflate`. Conflate can be thought as a special `reduce` operation that collapses multiple upstream elements into one aggregate element if needed to keep the speed of the upstream unaffected by the downstream.

When the upstream is faster, the reducing process of the `conflate` starts. Our reducer function simply takes the freshest element. This in a simple dropping operation.

```
1. val droppyStream: Flow[Message, Message, NotUsed] =
2.   Flow[Message].conflate((lastMessage, newMessage) => newMessage)
```

There is a more general version of `conflate` named `conflateWithSeed` that allows to express more complex aggregations, more similar to a `fold`.

## Dropping broadcast

**Situation:** The default `Broadcast` graph element is properly backpressured, but that means that a slow downstream consumer can hold back the other downstream consumers resulting in lowered throughput. In other words the rate of `Broadcast` is the rate of its slowest downstream consumer. In certain cases it is desirable to allow faster consumers to progress independently of their slower siblings by dropping elements if necessary.

One solution to this problem is to append a `buffer` element in front of all of the downstream consumers defining a dropping strategy instead of the default `Backpressure`. This allows small temporary rate differences between the different consumers (the buffer smooths out small rate variances), but also allows faster consumers to progress by dropping from the buffer of the slow consumers if necessary.

```
1. val graph = RunnableGraph.fromGraph(GraphDSL.create(mySink1, mySink2, mySink3)((_, _, _)) { implicit b => (sink1, sink2, sink3) =>
2.   import GraphDSL.Implicits._
3.
4.   val bcast = b.add(Broadcast[Int](3))
5.   myElements ~> bcast
```

```
6.
7.   bcast.buffer(10, OverflowStrategy.dropHead) ~> sink1
8.   bcast.buffer(10, OverflowStrategy.dropHead) ~> sink2
9.   bcast.buffer(10, OverflowStrategy.dropHead) ~> sink3
10.  ClosedShape
11. })
```

## Collecting missed ticks

**Situation:** Given a regular (stream) source of ticks, instead of trying to backpressure the producer of the ticks we want to keep a counter of the missed ticks instead and pass it down when possible.

We will use `conflateWithSeed` to solve the problem. The seed version of conflate takes two functions:

- A seed function that produces the zero element for the folding process that happens when the upstream is faster than the downstream. In our case the seed function is a constant function that returns 0 since there were no missed ticks at that point.
- A fold function that is invoked when multiple upstream messages needs to be collapsed to an aggregate value due to the insufficient processing rate of the downstream. Our folding function simply increments the currently stored count of the missed ticks so far.

As a result, we have a flow of `Int` where the number represents the missed ticks. A number 0 means that we were able to consume the tick fast enough (i.e. zero means: 1 non-missed tick + 0 missed ticks)

```
1. val missedTicks: Flow[Tick, Int, NotUsed] =
2.   Flow[Tick].conflateWithSeed(seed = (_) => 0)(
3.     (missedTicks, tick) => missedTicks + 1)
```

## Create a stream processor that repeats the last element seen

**Situation:** Given a producer and consumer, where the rate of neither is known in advance, we want to ensure that none of them is slowing down the other by dropping earlier unconsumed elements from the upstream if necessary, and repeating the last value for the downstream if necessary.

We have two options to implement this feature. In both cases we will use `GraphStage` to build our custom element. In the first version we will use a provided initial value `initial` that will be used to feed the downstream if no upstream element is ready yet. In the `onPush()` handler we just overwrite the `currentValue` variable and immediately relieve the upstream by calling `pull()`. The downstream `onPull` handler is very similar, we immediately relieve the downstream by emitting `currentValue`.

```
import akka.stream._
import akka.stream.stage._
final class HoldWithInitial[T](initial: T) extends GraphStage[FlowShape[T, T]] {
  val in = Inlet[T]("HoldWithInitial.in")
  val out = Outlet[T]("HoldWithInitial.out")

  override val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {
    private var currentValue: T = initial

    setHandlers(in, out, new InHandler with OutHandler {
      override def onPush(): Unit = {
        currentValue = grab(in)
        pull(in)
      }

      override def onPull(): Unit = {
        push(out, currentValue)
      }
    })

    override def preStart(): Unit = {
      pull(in)
    }
  }

}
```

While it is relatively simple, the drawback of the first version is that it needs an arbitrary initial element which is not always possible to provide. Hence, we create a second version where the downstream might need to wait in one single case: if the very first element is not yet available.

We introduce a boolean variable `waitingFirstValue` to denote whether the first element has been provided or not (alternatively an `Option` can be used for `currentValue` or if the element type is a subclass of AnyRef a null can be used with the same purpose). In the downstream `onPull()` handler the difference from the previous version is that we check if we have received the first value and only emit if we have. This leads to that when the first element comes in we must check if there possibly already was demand from downstream so that we in that case can push the element directly.

```
1. import akka.stream._
2. import akka.stream.stage._
3. final class HoldWithWait[T] extends GraphStage[FlowShape[T, T]] {
4.   val in = Inlet[T]("HoldWithWait.in")
5.   val out = Outlet[T]("HoldWithWait.out")
6.
7.   override val shape = FlowShape.of(in, out)
8.
9.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {
10.     private var currentValue: T = _
11.     private var waitingFirstValue = true
12.
13.     setHandlers(in, out, new InHandler with OutHandler {
14.       override def onPush(): Unit = {
15.         currentValue = grab(in)
16.         if (waitingFirstValue) {
17.           waitingFirstValue = false
18.           if (isAvailable(out)) push(out, currentValue)
19.         }
20.         pull(in)
21.       }
22.
23.       override def onPull(): Unit = {
24.         if (!waitingFirstValue) push(out, currentValue)
25.       }
26.     })
27.
28.     override def preStart(): Unit = {
29.       pull(in)
```

```
30.       }
31.     }
32.   }
```

## Globally limiting the rate of a set of streams

**Situation:** Given a set of independent streams that we cannot merge, we want to globally limit the aggregate throughput of the set of streams.

One possible solution uses a shared actor as the global limiter combined with mapAsync to create a reusable `Flow` that can be plugged into a stream to limit its rate.

As the first step we define an actor that will do the accounting for the global rate limit. The actor maintains a timer, a counter for pending permit tokens and a queue for possibly waiting participants. The actor has an `open` and `closed` state. The actor is in the `open` state while it has still pending permits. Whenever a request for permit arrives as a `WantToPass` message to the actor the number of available permits is decremented and we notify the sender that it can pass by answering with a `MayPass` message. If the amount of permits reaches zero, the actor transitions to the `closed` state. In this state requests are not immediately answered, instead the reference of the sender is added to a queue. Once the timer for replenishing the pending permits fires by sending a `ReplenishTokens` message, we increment the pending permits counter and send a reply to each of the waiting senders. If there are more waiting senders than permits available we will stay in the `closed` state.

```
1.  object Limiter {
2.    case object WantToPass
3.    case object MayPass
4.
5.    case object ReplenishTokens
6.
7.    def props(maxAvailableTokens: Int, tokenRefreshPeriod: FiniteDuration,
8.              tokenRefreshAmount: Int): Props =
9.      Props(new Limiter(maxAvailableTokens, tokenRefreshPeriod, tokenRefreshAmount))
10. }
11.
12. class Limiter(
13.   val maxAvailableTokens: Int,
14.   val tokenRefreshPeriod: FiniteDuration,
15.   val tokenRefreshAmount: Int) extends Actor {
16.   import Limiter._
```

```
17.   import context.dispatcher
18.   import akka.actor.Status
19.
20.   private var waitQueue = immutable.Queue.empty[ActorRef]
21.   private var permitTokens = maxAvailableTokens
22.   private val replenishTimer = system.scheduler.schedule(
23.     initialDelay = tokenRefreshPeriod,
24.     interval = tokenRefreshPeriod,
25.     receiver = self,
26.     ReplenishTokens)
27.
28.   override def receive: Receive = open
29.
30.   val open: Receive = {
31.     case ReplenishTokens =>
32.       permitTokens = math.min(permitTokens + tokenRefreshAmount, maxAvailableTokens)
33.     case WantToPass =>
34.       permitTokens -= 1
35.       sender() ! MayPass
36.       if (permitTokens == 0) context.become(closed)
37.   }
38.
39.   val closed: Receive = {
40.     case ReplenishTokens =>
41.       permitTokens = math.min(permitTokens + tokenRefreshAmount, maxAvailableTokens)
42.       releaseWaiting()
43.     case WantToPass =>
44.       waitQueue = waitQueue.enqueue(sender())
45.   }
46.
47.   private def releaseWaiting(): Unit = {
48.     val (toBeReleased, remainingQueue) = waitQueue.splitAt(permitTokens)
49.     waitQueue = remainingQueue
```

```
      permitTokens -= toBeReleased.size
      toBeReleased foreach (_ ! MayPass)
      if (permitTokens > 0) context.become(open)
    }

    override def postStop(): Unit = {
      replenishTimer.cancel()
      waitQueue foreach (_ ! Status.Failure(new IllegalStateException("limiter stopped")))
    }
  }
```

To create a Flow that uses this global limiter actor we use the `mapAsync` function with the combination of the `ask` pattern. We also define a timeout, so if a reply is not received during the configured maximum wait period the returned future from `ask` will fail, which will fail the corresponding stream as well.

```
def limitGlobal[T](limiter: ActorRef, maxAllowedWait: FiniteDuration): Flow[T, T, NotUsed] = {
  import akka.pattern.ask
  import akka.util.Timeout
  Flow[T].mapAsync(4)((element: T) => {
    import system.dispatcher
    implicit val triggerTimeout = Timeout(maxAllowedWait)
    val limiterTriggerFuture = limiter ? Limiter.WantToPass
    limiterTriggerFuture.map((_) => element)
  })

}
```

**Note**

The global actor used for limiting introduces a global bottleneck. You might want to assign a dedicated dispatcher for this actor.

# Working with IO

## Chunking up a stream of ByteStrings into limited size ByteStrings

**Situation:** Given a stream of ByteStrings we want to produce a stream of ByteStrings containing the same bytes in the same sequence, but capping the size of ByteStrings. In other words we want to slice up ByteStrings into smaller chunks if they exceed a size threshold.

This can be achieved with a single `GraphStage`. The main logic of our stage is in `emitChunk()` which implements the following logic:

- if the buffer is empty, and upstream is not closed we pull for more bytes, if it is closed we complete
- if the buffer is nonEmpty, we split it according to the `chunkSize`. This will give a next chunk that we will emit, and an empty or nonempty remaining buffer.

Both `onPush()` and `onPull()` calls `emitChunk()` the only difference is that the push handler also stores the incoming chunk by appending to the end of the buffer.

```
1. import akka.stream.stage._
2.
3. class Chunker(val chunkSize: Int) extends GraphStage[FlowShape[ByteString, ByteString]] {
4.   val in = Inlet[ByteString]("Chunker.in")
5.   val out = Outlet[ByteString]("Chunker.out")
6.   override val shape = FlowShape.of(in, out)
7.
8.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {
9.     private var buffer = ByteString.empty
10.
11.    setHandler(out, new OutHandler {
12.      override def onPull(): Unit = {
13.        if (isClosed(in)) emitChunk()
14.        else pull(in)
15.      }
16.    })
17.    setHandler(in, new InHandler {
18.      override def onPush(): Unit = {
19.        val elem = grab(in)
20.        buffer ++= elem
```

```
21.         emitChunk()
22.       }
23.
24.       override def onUpstreamFinish(): Unit = {
25.         if (buffer.isEmpty) completeStage()
26.         else {
27.           // There are elements left in buffer, so
28.           // we keep accepting downstream pulls and push from buffer until emptied.
29.           //
30.           // It might be though, that the upstream finished while it was pulled, in which
31.           // case we will not get an onPull from the downstream, because we already had one.
32.           // In that case we need to emit from the buffer.
33.           if (isAvailable(out)) emitChunk()
34.         }
35.       }
36.     })
37.
38.     private def emitChunk(): Unit = {
39.       if (buffer.isEmpty) {
40.         if (isClosed(in)) completeStage()
41.         else pull(in)
42.       } else {
43.         val (chunk, nextBuffer) = buffer.splitAt(chunkSize)
44.         buffer = nextBuffer
45.         push(out, chunk)
46.       }
47.     }
48.
49.   }
50. }
51.
52. val chunksStream = rawBytes.via(new Chunker(ChunkLimit))
```

## Limit the number of bytes passing through a stream of ByteStrings

**Situation:** Given a stream of ByteStrings we want to fail the stream if more than a given maximum of bytes has been consumed.

This recipe uses a `GraphStage` to implement the desired feature. In the only handler we override, `onPush()` we just update a counter and see if it gets larger than `maximumBytes`. If a violation happens we signal failure, otherwise we forward the chunk we have received.

```
1. import akka.stream.stage._
2. class ByteLimiter(val maximumBytes: Long) extends GraphStage[FlowShape[ByteString, ByteString]] {
3.   val in = Inlet[ByteString]("ByteLimiter.in")
4.   val out = Outlet[ByteString]("ByteLimiter.out")
5.   override val shape = FlowShape.of(in, out)
6.
7.   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {
8.     private var count = 0
9.
10.    setHandlers(in, out, new InHandler with OutHandler {
11.
12.      override def onPull(): Unit = {
13.        pull(in)
14.      }
15.
16.      override def onPush(): Unit = {
17.        val chunk = grab(in)
18.        count += chunk.size
19.        if (count > maximumBytes) failStage(new IllegalStateException("Too much bytes"))
20.        else push(out, chunk)
21.      }
22.    })
23.  }
24. }
25.
26. val limiter = Flow[ByteString].via(new ByteLimiter(SizeLimit))
```

### Compact ByteStrings in a stream of ByteStrings

**Situation:** After a long stream of transformations, due to their immutable, structural sharing nature ByteStrings may refer to multiple original ByteString instances unnecessarily retaining memory. As the final step of a transformation chain we want to have clean copies that are no longer referencing the original ByteStrings.

The recipe is a simple use of map, calling the `compact()` method of the `ByteString` elements. This does copying of the underlying arrays, so this should be the last element of a long chain if used.

```
1. val compacted: Source[ByteString, NotUsed] = data.map(_.compact)
```

### Injecting keep-alive messages into a stream of ByteStrings

**Situation:** Given a communication channel expressed as a stream of ByteStrings we want to inject keep-alive messages but only if this does not interfere with normal traffic.

There is a built-in operation that allows to do this directly:

```
1. import scala.concurrent.duration._
2. val injectKeepAlive: Flow[ByteString, ByteString, NotUsed] =
3.   Flow[ByteString].keepAlive(1.second, () => keepaliveMessage)
```

# Configuration

```
1. ####################################
2. # Akka Stream Reference Config File #
3. ####################################
4.
5. akka {
6.   stream {
7.
8.     # Default flow materializer settings
9.     materializer {
10.
11.       # Initial size of buffers used in stream elements
```

```
12.       initial-input-buffer-size = 4
13.       # Maximum size of buffers used in stream elements
14.       max-input-buffer-size = 16
15.
16.       # Fully qualified config path which holds the dispatcher configuration
17.       # to be used by FlowMaterialiser when creating Actors.
18.       # When this value is left empty, the default-dispatcher will be used.
19.       dispatcher = ""
20.
21.       # Cleanup leaked publishers and subscribers when they are not used within a given
22.       # deadline
23.       subscription-timeout {
24.         # when the subscription timeout is reached one of the following strategies on
25.         # the "stale" publisher:
26.         # cancel - cancel it (via `onError` or subscribing to the publisher and
27.         #          `cancel()`ing the subscription right away
28.         # warn   - log a warning statement about the stale element (then drop the
29.         #          reference to it)
30.         # noop   - do nothing (not recommended)
31.         mode = cancel
32.
33.         # time after which a subscriber / publisher is considered stale and eligible
34.         # for cancelation (see `akka.stream.subscription-timeout.mode`)
35.         timeout = 5s
36.       }
37.
38.       # Enable additional troubleshooting logging at DEBUG log level
39.       debug-logging = off
40.
41.       # Maximum number of elements emitted in batch if downstream signals large demand
42.       output-burst-limit = 1000
43.
44.       # Enable automatic fusing of all graphs that are run. For short-lived streams
```

```
45.       # this may cause an initial runtime overhead, but most of the time fusing is
46.       # desirable since it reduces the number of Actors that are created.
47.       auto-fusing = on
48.
49.       # Those stream elements which have explicit buffers (like mapAsync, mapAsyncUnordered,
50.       # buffer, flatMapMerge, Source.actorRef, Source.queue, etc.) will preallocate a fixed
51.       # buffer upon stream materialization if the requested buffer size is less than this
52.       # configuration parameter. The default is very high because failing early is better
53.       # than failing under load.
54.       #
55.       # Buffers sized larger than this will dynamically grow/shrink and consume more memory
56.       # per element than the fixed size buffers.
57.       max-fixed-buffer-size = 1000000000
58.
59.       # Maximum number of sync messages that actor can process for stream to substream communication.
60.       # Parameter allows to interrupt synchronous processing to get upsteam/downstream messages.
61.       # Allows to accelerate message processing that happening withing same actor but keep system responsive.
62.       sync-processing-limit = 1000
63.
64.       debug {
65.         # Enables the fuzzing mode which increases the chance of race conditions
66.         # by aggressively reordering events and making certain operations more
67.         # concurrent than usual.
68.         # This setting is for testing purposes, NEVER enable this in a production
69.         # environment!
70.         # To get the best results, try combining this setting with a throughput
71.         # of 1 on the corresponding dispatchers.
72.         fuzzing-mode = off
73.       }
74.     }
75.
76.     # Fully qualified config path which holds the dispatcher configuration
77.     # to be used by FlowMaterialiser when creating Actors for IO operations,
```

```
    # such as FileSource, FileSink and others.
    blocking-io-dispatcher = "akka.stream.default-blocking-io-dispatcher"

    default-blocking-io-dispatcher {
      type = "Dispatcher"
      executor = "thread-pool-executor"
      throughput = 1

      thread-pool-executor {
        core-pool-size-min = 2
        core-pool-size-factor = 2.0
        core-pool-size-max = 16
      }
    }
  }

  # configure overrides to ssl-configuration here (to be used by akka-streams, and akka-http - i.e. when serving https connections)
  ssl-config {
    protocol = "TLSv1.2"
  }
}
```

# Інструкція з міграції з 2.0.x до 2.4.x

## Загальні зауваження

### akka.Done та akka.NotUsed замінили Unit та BoxedUnit

Щоб провадити більш ясні сігнатури, та мати уніфікований API для обох, Java та Scala, були введені два нові типи:

`akka.NotUsed` призначений бути використаний замість `Unit` в Scala та `BoxedUnit` в Java, щоб позначити. що тип параметра необзідний, але не використовується. Це загальний випадок для `Source`, `Flow` та `Sink`, що не матеріалізують жодного значення.

`akka.Done` доданий для випадка використання, коли він замкнений в іншому об'єкті, щоб позначити завершення, але немає справжнього значення, приєднаного до завершення. Він використовується для заміни входжень `Future<BoxedUnit>` на `Future<Done>` в Java, та `Future[Unit]` на `Future[Done]` в Scala.

Всі попередні використання `Unit` та `BoxedUnit` для цих двох випадків в Akka Streams API біли оновлені.

Це означає, що код Scala як цей:

```
1. Source[Int, Unit] source = Source.from(1 to 5)
2. Sink[Int, Future[Unit]] sink = Sink.ignore()
```

потрібно змінити на наступний:

```
1. Source[Int, NotUsed] source = Source.from(1 to 5)
2. Sink[Int, Future[Done]] sink = Sink.ignore()
```

Ці зміни стосуються до всіх місць, де використовуються потоки, що означає, що сигнатури API стійких запитів також зазнали змін.

# Видалено ImplicitMaterializer

Допоміжний трейт `ImplicitMaterializer` був видалений, оскільки його було важко знайти, та можливість не була не варта додаткового трейта. Визначення неявного матеріалізатора в оточуючому акторові може бути зроблена таким чином:

```
1. final implicit val materializer: ActorMaterializer = ActorMaterializer(ActorMaterializerSettings(context.system))
```

# Змінені операції

## `expand()` тепер базується на Iterator

Перед цім комбінатор `expand` потребував двох вхідних функцій: перша підіймала входящі значення в екстрапольованому стані, та друга виділяла з цього значення, можливо включаючи цей стан. Це було спрощено в єдину функцію, що перетворює входящий елемент в Iterator.

Найбільш помітний випадок використання раніше було просто повторювати отримане перед цім значення:

```
1. Flow[Int].expand(identity)(s => (s, s)) // Це більше на працює!
```

В Akka 2.4.x це спрощене до:

```
1. Flow[Int].expand(Iterator.continually(_))
```

Якщо стан вимагає утримуватись на під час процеса розширення, тоді цей стан буде потребувати бути обробленим в ітераторі. Приклад підрахунку числа розширень раніше міг виглядати так:

```
1. // Це більше не працює!
2. Flow[Int].expand((_, 0)){ case (in, count) => (in, count) -> (in, count + 1) }
```

В Akka 2.4.x це формулюється таким чином:

```
1. Flow[Int].expand(i => {
2.   var state = 0
3.   Iterator.continually({
4.     state += 1
5.     (i, state)
6.   })
7. })
```

## `conflate` був переіменований на `conflateWithSeed()`

Новий оператор `conflate` є спеціальним випадком оригінальної поведінки (переіменований на `conflateWithSeed`), що не змінює тип потока. Використання нового оператора таке просте, як:

```
1. Flow[Int].conflate(_ + _) // Додає числа, доки даунстрім не готовий
```

Що те ж саме, що використовувати `conflateWithSeed` з функцією identity

> Flow[Int].conflateWithSeed(identity)(_ + _) // Додавати числа, доки даунстрім не готовий

### `viaAsync` та `viaAsyncMat` були замінені на `async`

`async` доступний з `Sink`, `Source`, `Flow` потоків-заглушок. Він провадить скорочення для встановлення атрибута `Attributes.asyncBoundary` на потоці. Існуючі методи `Flow.viaAsync` та `Flow.viaAsyncMat` були видалені, щоб зробити маркування асинхронних меж більш узгодженими:

```
1. // Це більше не працює
2. source.viaAsync(flow)
```

В Akka 2.4.x це буде замість цього виглядати так:

```
1. val flow = Flow[Int].map(_ + 1)
2. Source(1 to 10).via(flow.async)
```

## Зміни в Akka HTTP

### Ім'я параметра налаштування маршрутизації

`RoutingSettings` були до цього єдиним налаштуванням, доступним на `RequestContext`, та були доступні через `settings`. Тепер ми зробили можливим сконфігурувати також налаштування парсерів, так що `RoutingSettings` тепер `routingSettings` та `ParserSettings` тепер доступні через `parserSettings`.

### Поведінка клієнт/сервер на перерваній сутності

Раніше, якщо запит або відповідь були перервані або спожиті тільки частково (тобто з використанням комбінатора `take`), залишок даних був тихо поглинутий для запобігання звалювання з'єднання, оскільки можуть бути більше надходжень запитів/відповідей. Тепер поведінка по замовчанню є закриття з'єднання, щоб запобігти надмірного використання ресурсів в випадку величезних сутностей.

Стара поведінка може бути досягнена через примусове очищення сутностей:

> response.entity.dataBytes.runWith(Sink.ignore)

# Змінені  Source/Sink

## IO Source/Sink матеріалізують IOResult

Матеріалізовані значення наступних джерел та приймачів:

- `FileIO.fromPath`
- `FileIO.toPath`
- `StreamConverters.fromInputStream`
- `StreamConverters.fromOutputStream`

були змінені з `Long` на `akka.stream.io.IOResult`. Це дозволяє надсилати більш складні сигнали сценаріїв завершення. Наприклад, по збою тепер можливо повертати виключення, та число байт, записаних до того, як виникло виключення.

# PushStage, PushPullStage та DetachedStage пішли в минуле на користь GraphStage

Класи `PushStage` `PushPullStage` та `DetachedStage` пішли у відставку, та мають бути замінені на `GraphStage` (Власна обробка з використанням GraphStage), що тепер є єдиним потужним API для власної обробки потоків.

## Процедура оновелння

Будь ласка, проконсультуйтесь з документацією `GraphStage` (Власна обробка за допомогою GraphStage) та попередньою інструкцією по міграції щодо міграції `AsyncStage` на `GraphStage`.

## Websocket тепер узгоджено названий WebSocket

Попередньо ми мали суміш методів та класів, названих `websocket` або `Websocket`, що будо в супереч з тим, як промовляється це слово в специфікації та деяких інших місцях Akka HTTP.

Методи та класи, що використовували слово WebSocket, тепер узгоджено використовують його як `WebSocket`, так що оновлення є простим як знайти-та-замінити малу `s` на велику `S`, кожного разу, коли трапляється.

## Java DSL для прив'язок та з'єднань Http змінено

Щоб мінімізувати число потрібний перевантажень для кожного метода, визначеного на `Http` розширенні, був введений новий міні-DSL для з'єднання до вузлів на основі імена вузла, порта та опціонального `ConnectionContext`.

Доступність контексту з'єднання (якщо він встановлений до `HttpsConnectionContext`) робить сервер прив'язаним як HTTPS сервер, та для вихідних з'єднань ці налаштування використовуються замість налаштувань по замовчанню, якщо надані.

Було:

```
1. http.cachedHostConnectionPool(toHost("akka.io"), materializer());
2. http.cachedHostConnectionPool("akka.io", 80, httpsConnectionContext, materializer()); // does not work anymore
```

Замінене на:

```
1. http.cachedHostConnectionPool(toHostHttps("akka.io", 8081), materializer());
2. http.cachedHostConnectionPool(toHostHttps("akka.io", 8081).withCustomHttpsContext(httpsContext), materializer());
```

## SslTls був переіменований на TLS та переміщений

DSL для доступу до TLS (або SSL) `BidiFlow` тепер розділений між пакунками `javadsl` та `scaladsl`, та був переіменований на `TLS`. Загальні опціональні типи (режими закриття, режими аутентифікації, тощо) були переміщені на верхній рівень пакунка `stream`, та загальні типи повідомлень доступні в класі `akka.stream.TLSProtocol`

## Фрейми переміщені до akka.stream.[javadsl/scaladsl]

Об'єкт `Framing`, що може використовуватись для нарізання потоків `ByteString` в шматки, залежні від фреймів (такі, як рядки) був переміщений до `akka.stream.scaladsl.Framing`, та отримав еквівалент Java DSL в вигляді `akka.stream.javadsl.Framing`.