REAL WORLD

AKKA STREAM

About Me

知乎
www.zhihu.com

罗辑　*Search Tech Lead*

Fall in love with scala since 2014

知乎
www.zhihu.com

Agenda

- Reactive Streams
- Akka Stream
- Akka Stream in Action
- Demo!

知乎
www.zhihu.com

What is a stream?

1. Ephemeral flow of data

2. Possibly unbounded in size

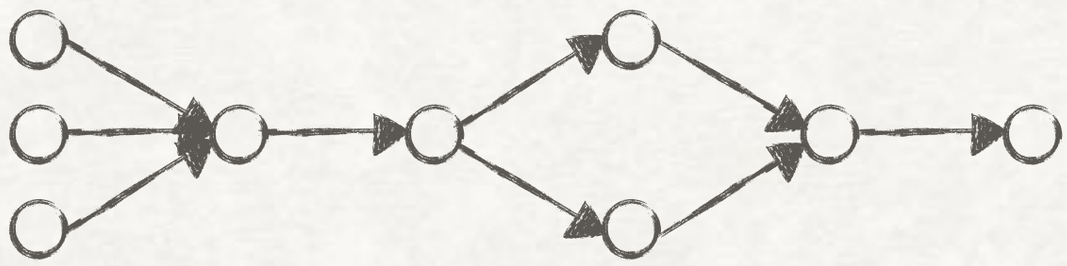
3. Focused on describing transformation

4. Can be formed into processing networks

知乎
www.zhihu.com

What is a stream?

1. Ephemeral flow of data
2. Possibly unbounded in size
3. Focused on describing transformation
4. Can be formed into processing networks



知乎
www.zhihu.com

Reactive Streams

“

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

http://www.reactive-streams.org

”

知乎
www.zhihu.com

“

Reactive Streams is an initiative to provide a **standard** for asynchronous stream processing with non-blocking back pressure.

http://www.reactive-streams.org

”

Reactive Streams API

知乎
www.zhihu.com

```
trait Publisher[T] {
  def subscribe(sub: Subscriber[T]): Unit
}

trait Subscription {
  def request(n: Int): Unit
  def cancel(): Unit
}

trait Subscriber[T] {
  def onSubscribe(s: Subscription): Unit
  def onNext(e: T): Unit
  def onError(t: Throwable): Unit
  def onComplete(): Unit
}
```

知乎
www.zhihu.com

Reactive Streams Implementations

rxjava @Netflix

reactor @SpringSource

vert.x @RedHat

akka-stream @Lightbend(Typesafe)

“

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking **back pressure**.

http://www.reactive-streams.org

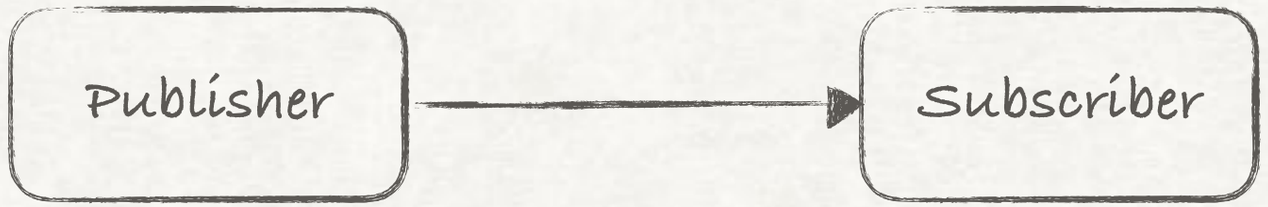
”

知乎
www.zhihu.com

Problem?

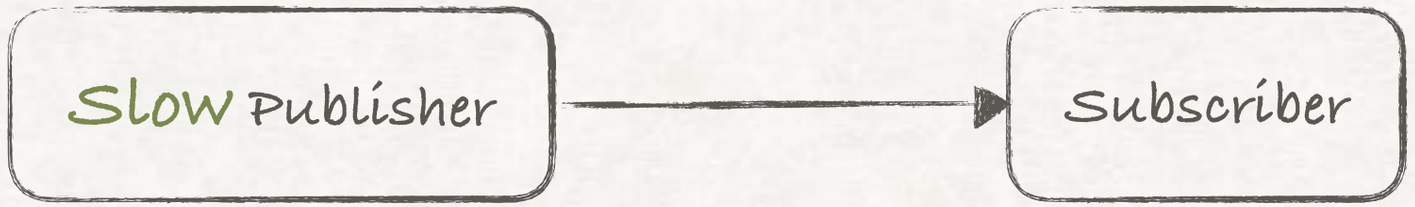
Reactive Streams

知乎
www.zhihu.com



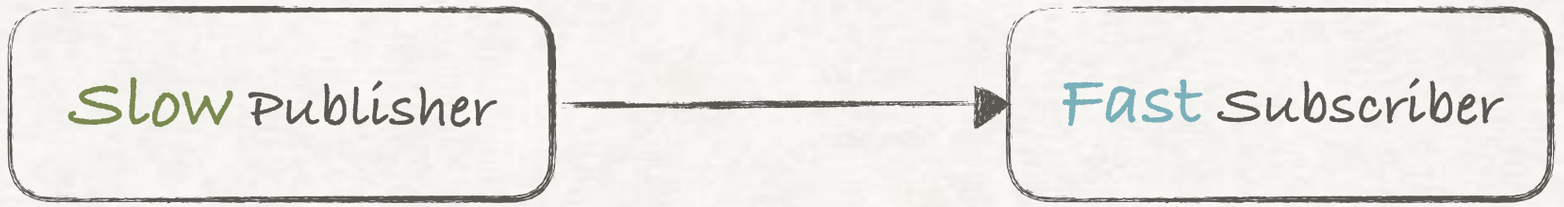
Reactive Streams

知乎
www.zhihu.com



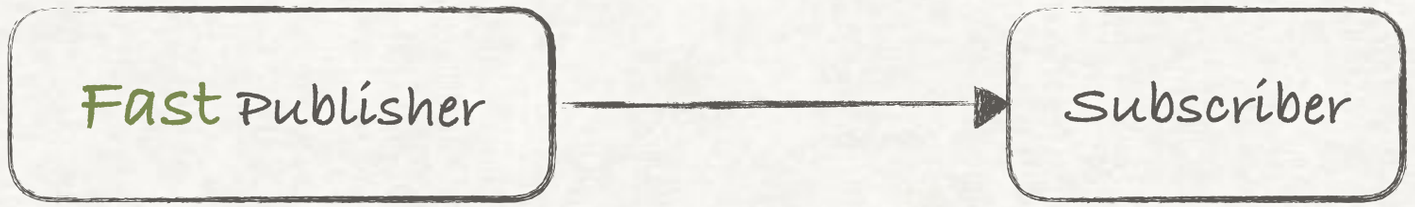
Reactive Streams

知乎
www.zhihu.com



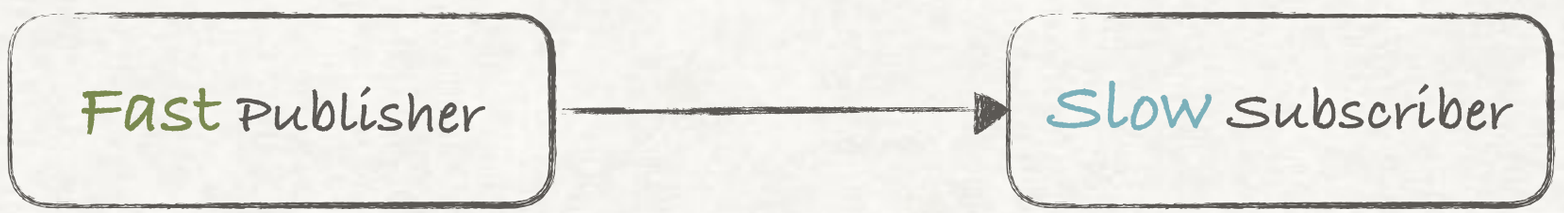
Reactive Streams

知乎
www.zhihu.com

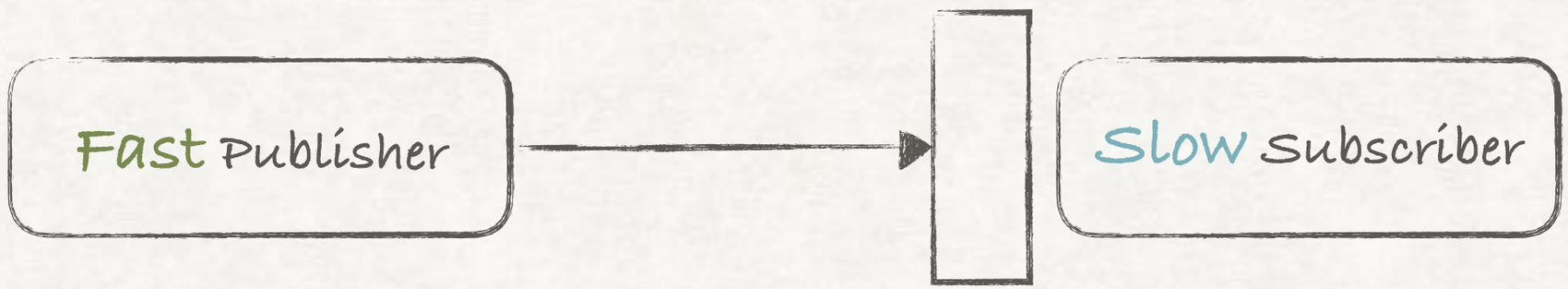


Reactive Streams

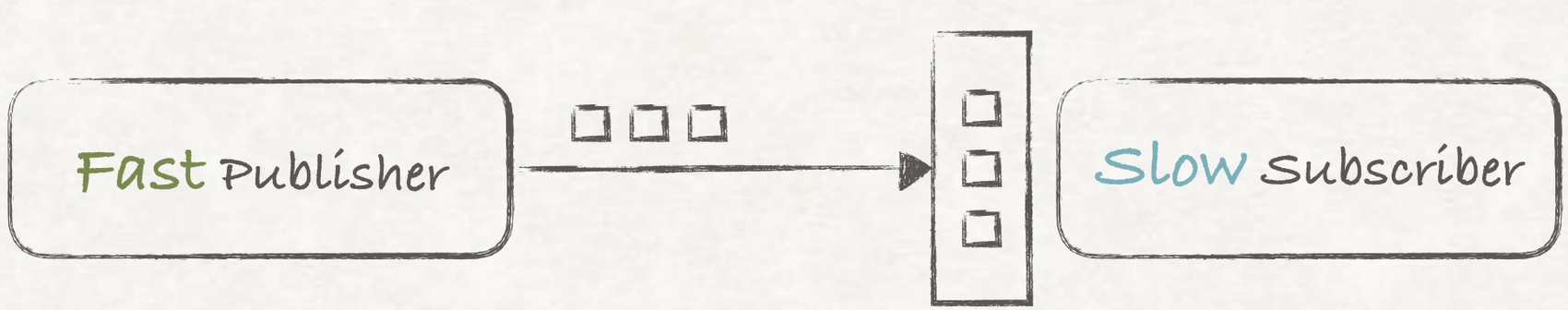
知乎
www.zhihu.com



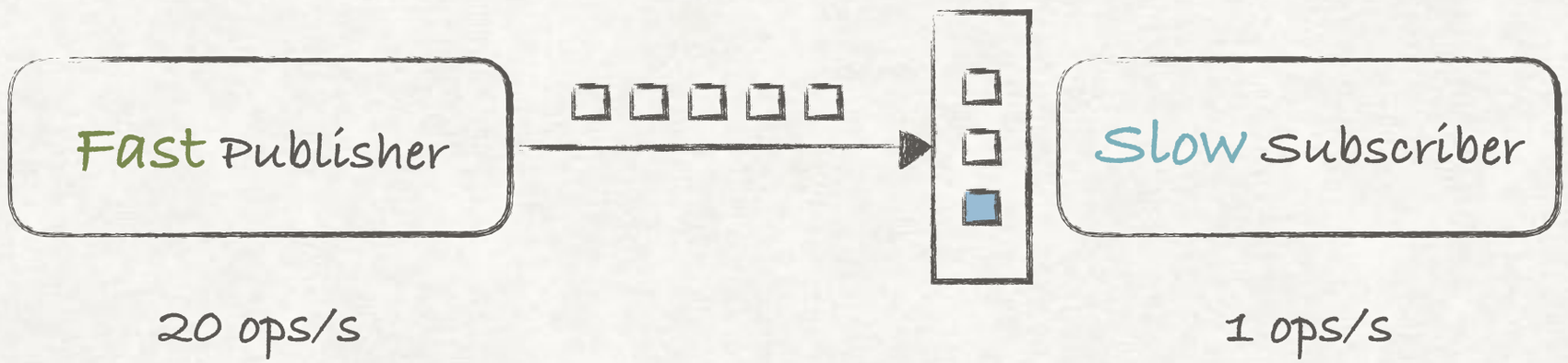
Use bounded buffer



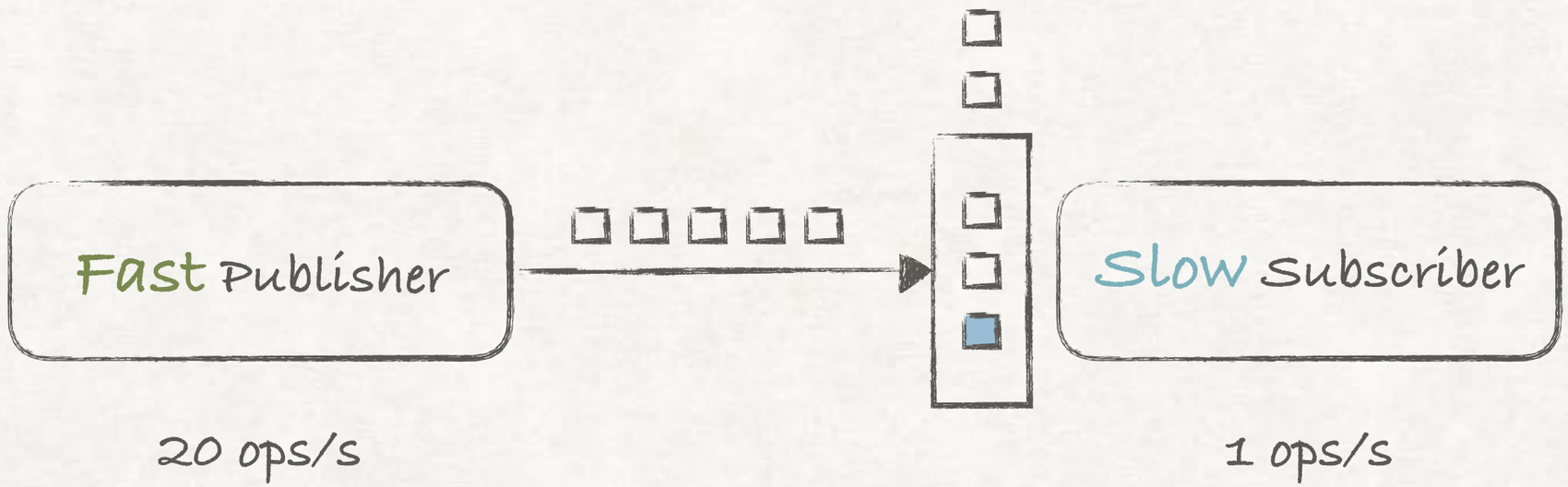
Use bounded buffer



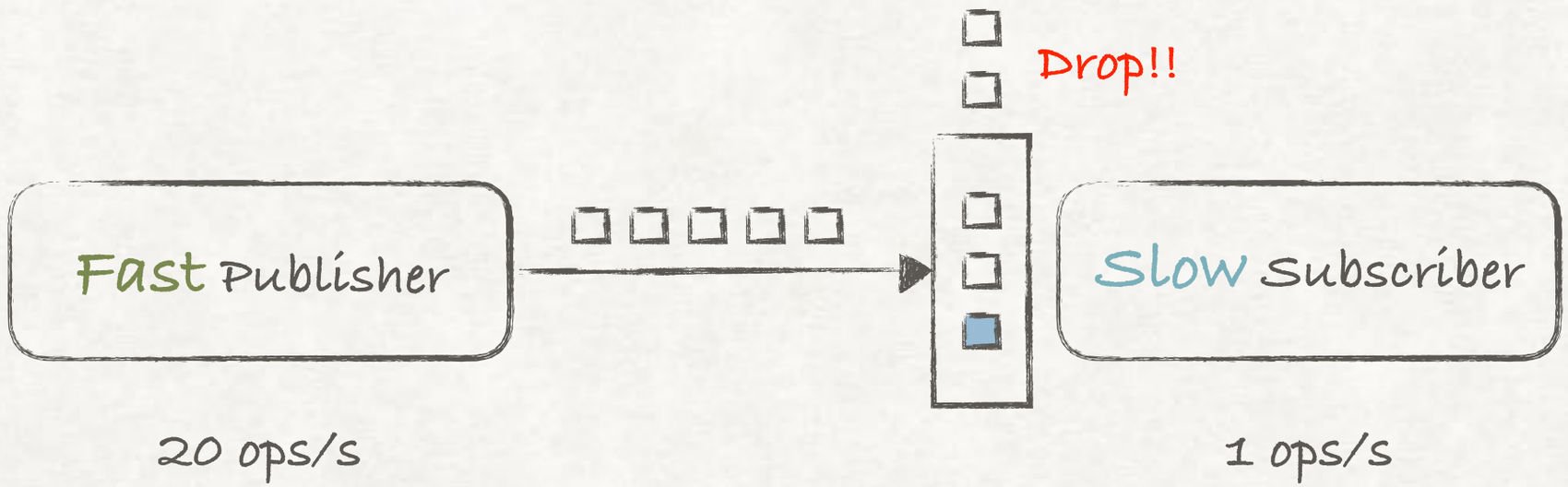
Use bounded buffer



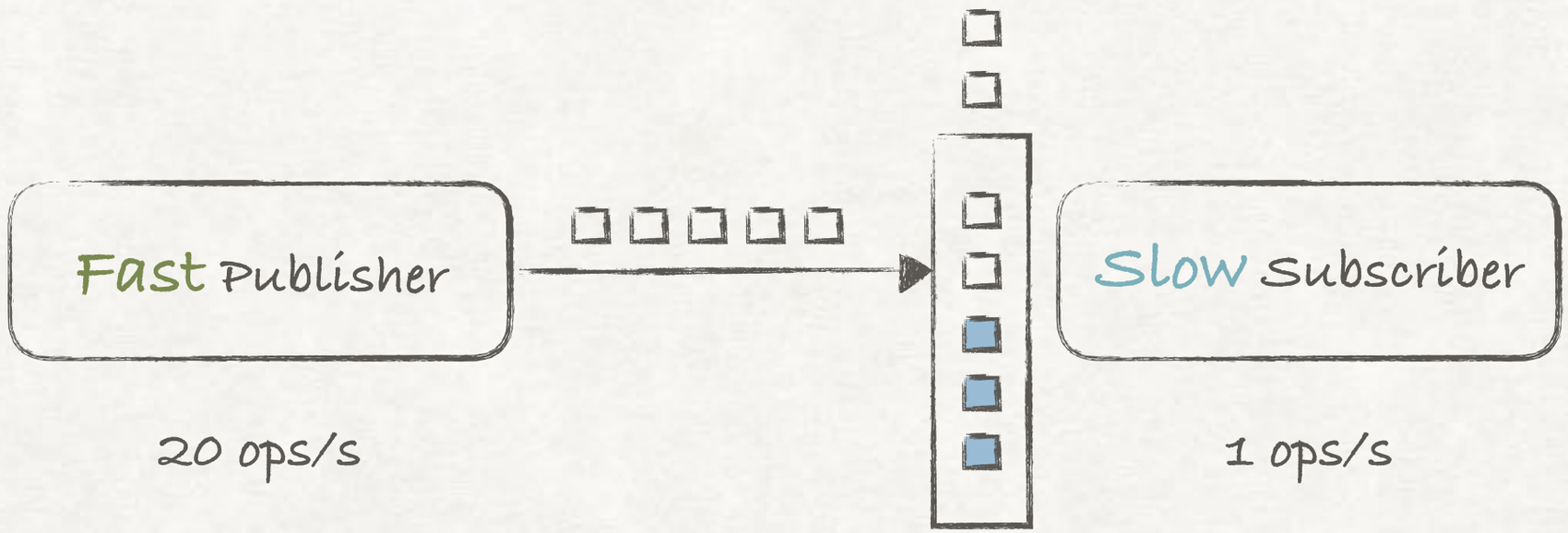
Use bounded buffer



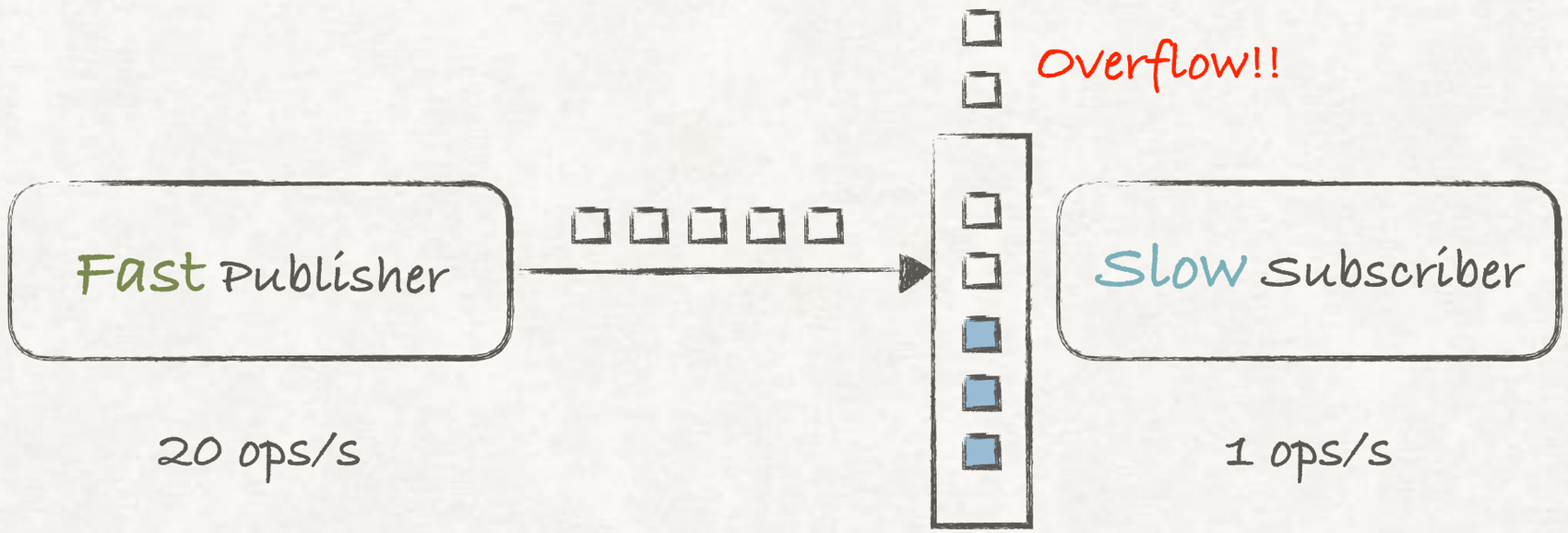
Use bounded buffer



Increase buffer size until memory exhausted…



Increase buffer size until memory exhausted…



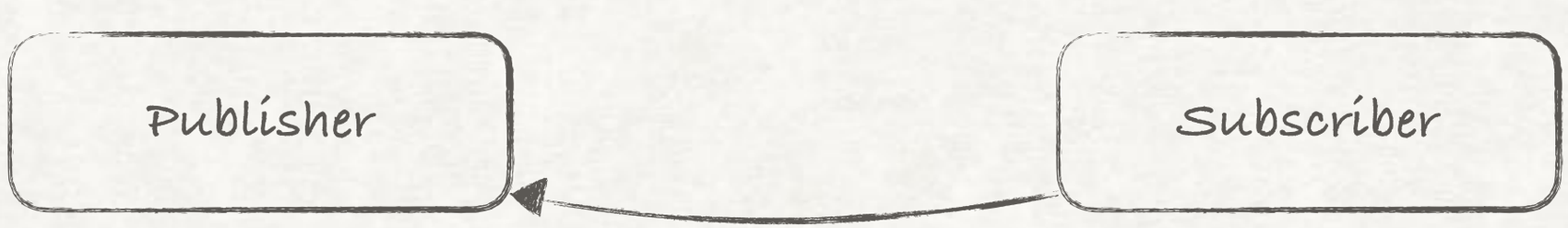
知乎
www.zhihu.com

What is back pressure?

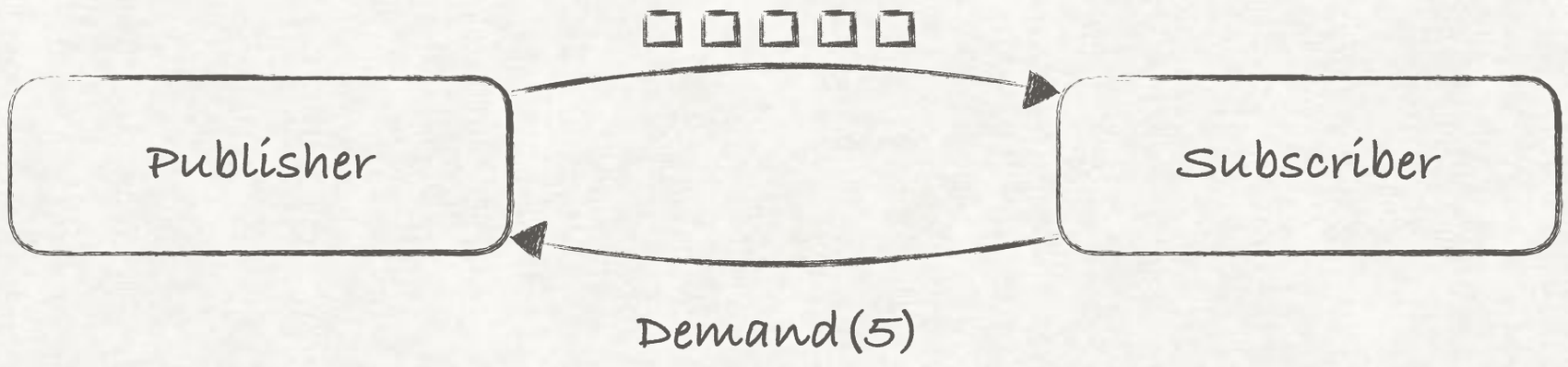
知乎
www.zhihu.com

Back pressure = Dynamic push/pull

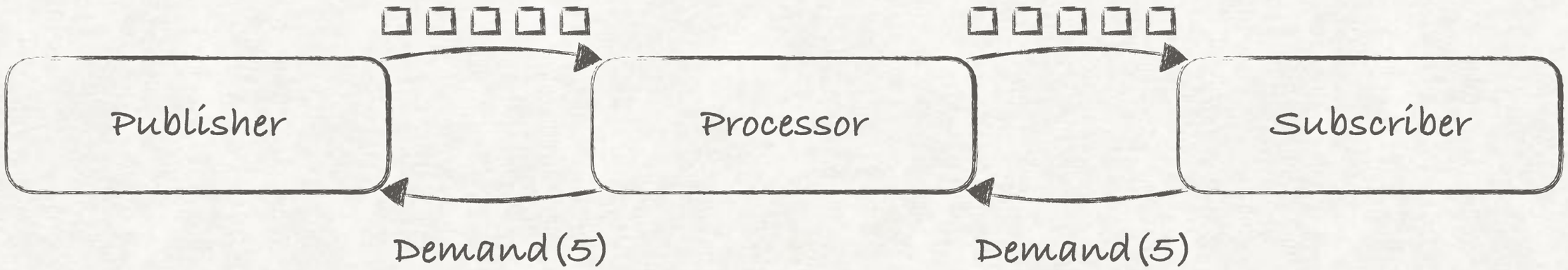
Subscriber demands 5 elements



Publisher sends at-most 5 elements.



Back pressure is **contagious**



Back pressure is **contagious**

Subscriber is slow ~> **Processor** must slow down ~> **Publisher** must slow down

Publisher back pressure strategies

Publisher back pressure strategies

1. Not generate elements, if it is able to **control** their production rate

Publisher back pressure strategies

1. Not generate elements, if it is able to **control** their production rate

2. Try **buffering** the elements in a bounded manner until more demand is signaled

Publisher back pressure strategies

1. Not generate elements, if it is able to **control** their production rate

2. Try **buffering** the elements in a bounded manner until more demand is signaled

3. **Drop** elements until more demand is signaled

Publisher back pressure strategies

1. Not generate elements, if it is able to **control** their production rate

2. Try **buffering** the elements in a bounded manner until more demand is signaled

3. **Drop** elements until more demand is signaled

4. **Tear down** the stream if unable to apply any of the above strategies

知乎
www.zhihu.com

Akka Stream

Akka Stream in 20 seconds

知乎
www.zhihu.com

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

Source(1 to 5).map(_.toString).runForeach(println)
```

Akka Stream in 20 seconds

知乎
www.zhihu.com

```scala
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val source = Source(1 to 5)

val flow = Flow[Int].map(_.toString)

val sink = Sink.foreach[String](println)

val runnable = source.via(flow).to(sink)

runnable.run()
```

Akka Stream in 20 seconds

知乎
www.zhihu.com

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val source = Source(1 to 5)

val flow = Flow[Int].map(_.toString)

val sink = Sink.foreach[String](println)

val runnable = source.via(flow).to(sink)

runnable.run()
```

Akka Stream in 20 seconds

知乎
www.zhihu.com

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val source = Source(1 to 5)

val flow = Flow[Int].map(_.toString)

val sink = Sink.foreach[String](println)

val runnable = source.via(flow).to(sink)

runnable.run()
```

Akka Stream in 20 seconds

知乎
www.zhihu.com

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val source = Source(1 to 5)

val flow = Flow[Int].map(_.toString)

val sink = Sink.foreach[String](println)

val runnable = source.via(flow).to(sink)

runnable.run()
```

Akka Stream in 20 seconds

知乎
www.zhihu.com

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val source = Source(1 to 5)

val flow = Flow[Int].map(_.toString)

val sink = Sink.foreach[String](println)

val runnable = source.via(flow).to(sink)

runnable.run()
```

Source *(Publisher)*

A processing stage with exactly **one output**, emitting data elements whenever downstream processing stages are ready to receive them.

Sink *(Subscriber)*

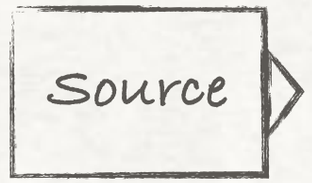
A processing stage with exactly **one input**, requesting and accepting data elements possibly slowing down the upstream producer of elements.

Flow *(Processor)*

A processing stage which has exactly **one input and output**, which connects its up- and downstreams by transforming the data elements flowing through it.

Source *(Publisher)*

A processing stage with exactly **one output**, emitting data elements whenever downstream processing stages are ready to receive them.



Sink *(Subscriber)*

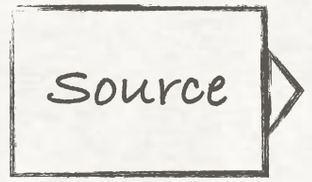
A processing stage with exactly **one input**, requesting and accepting data elements possibly slowing down the upstream producer of elements.

Flow *(Processor)*

A processing stage which has exactly **one input and output**, which connects its up- and downstreams by transforming the data elements flowing through it.

Source *(Publisher)*

A processing stage with exactly **one output**, emitting data elements whenever downstream processing stages are ready to receive them.



Sink *(Subscriber)*

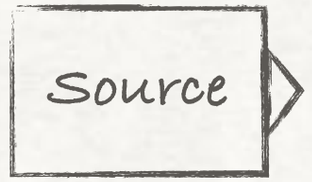
A processing stage with exactly **one input**, requesting and accepting data elements possibly slowing down the upstream producer of elements.

Flow *(Processor)*

A processing stage which has exactly **one input and output**, which connects its up- and downstreams by transforming the data elements flowing through it.

Source *(Publisher)*

A processing stage with exactly **one output**, emitting data elements whenever downstream processing stages are ready to receive them.

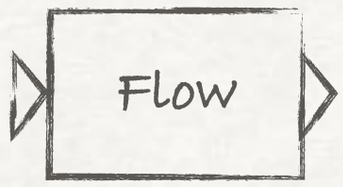


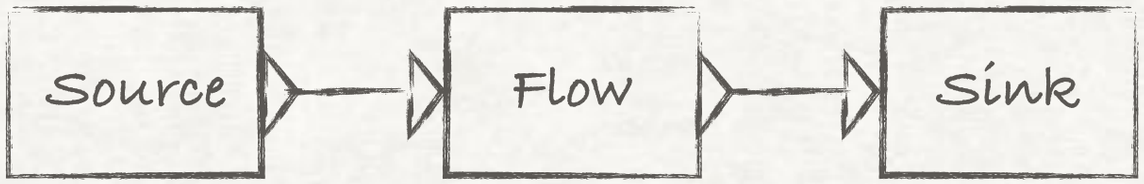
Sink *(Subscriber)*

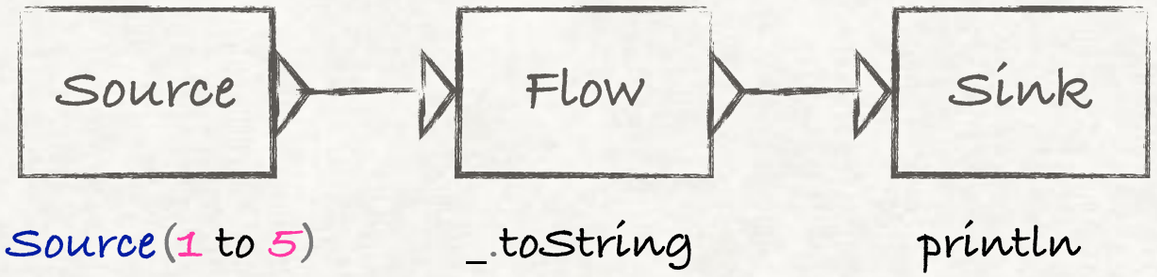
A processing stage with exactly **one input**, requesting and accepting data elements possibly slowing down the upstream producer of elements.

Flow *(Processor)*

A processing stage which has exactly **one input and output**, which connects its up- and downstreams by transforming the data elements flowing through it.

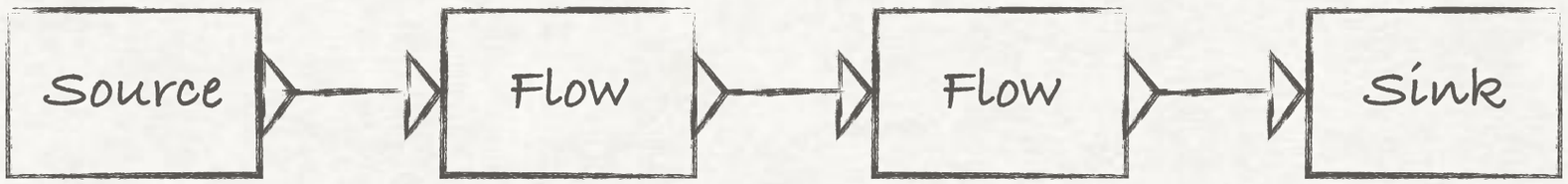




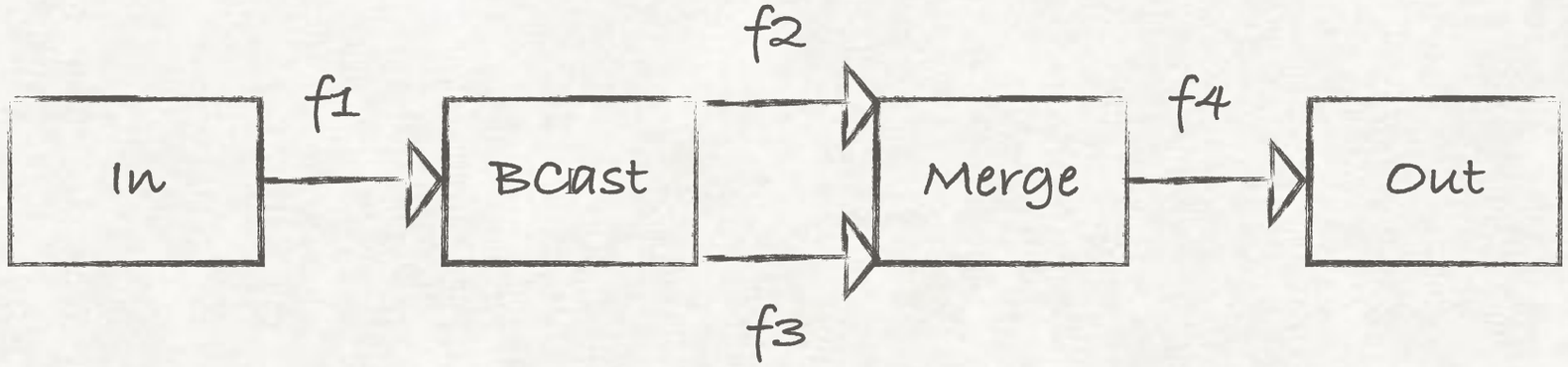


Source(1 to 5)　　_.toString　　println

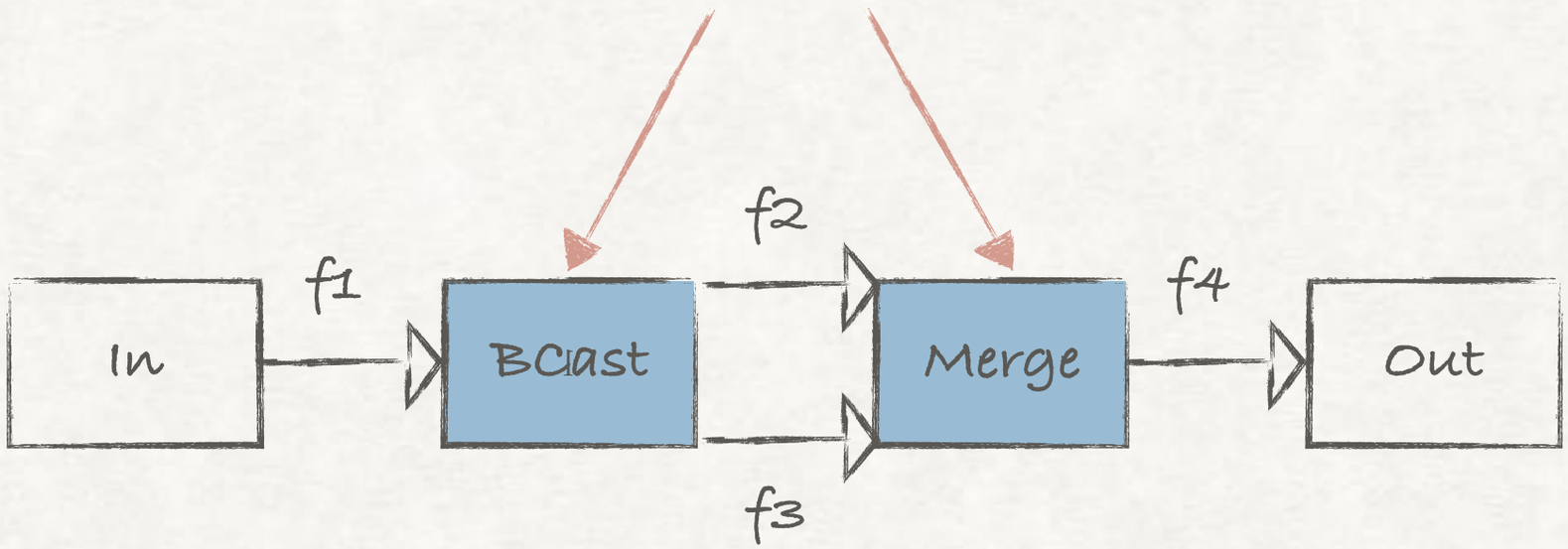
Linear Flow

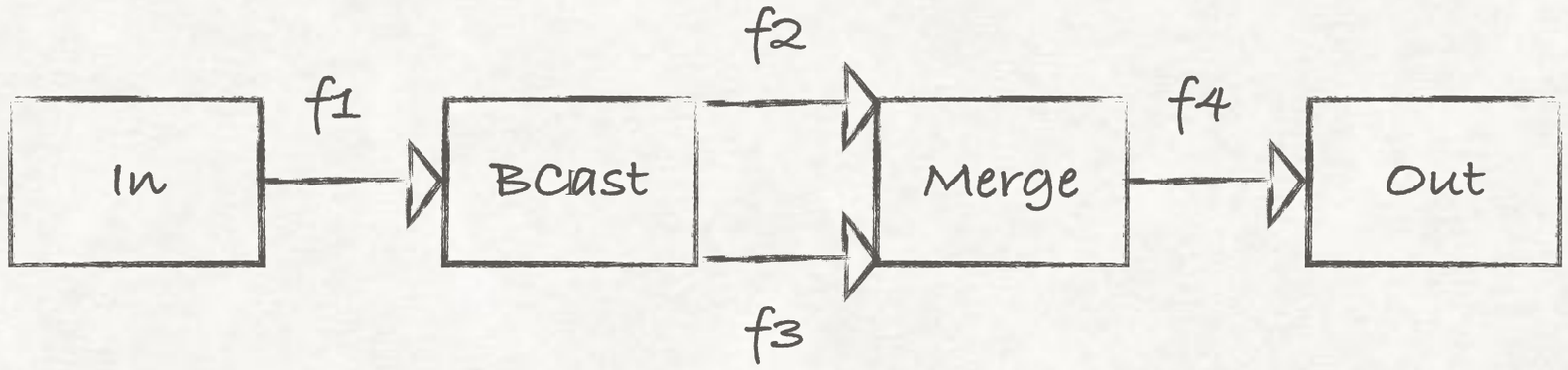


Graph Flow



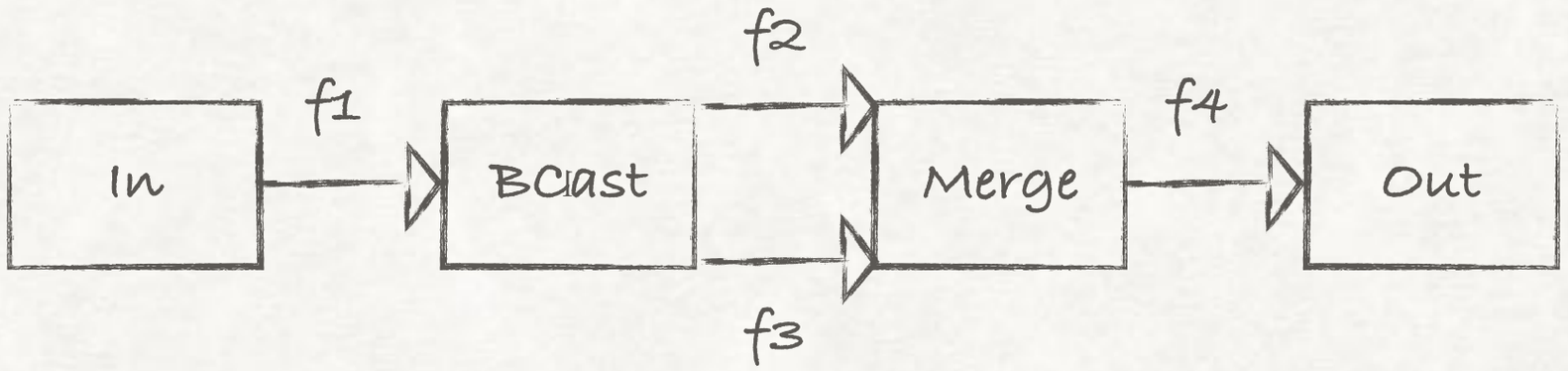
Fan-out & Fan-in






```
val f1 = Flow[Input].map(_.toIntermediate)
val f2 = Flow[Intermediate].map(_.enrich)
val f3 = Flow[Intermediate].mapFuture(_.enrichAsync)
val f4 = Flow[Enrich].filter(_.isImportant)

val in = SubscriberSource[Input]
val out = PublisherSink[Enrich]
```




```
val graph = Flow.fromGraph(GraphDSL.create() { implicit b =>
  val bcast = b.add(Broadcast[Intermediate](2))
  val merge = b.add(Merge[Enrich](2))

  in ~> f1 ~> bcast ~> f2 ~> merge
              bcast ~> f3 ~> merge ~> f4 ~> out

  FlowShape(bcast.in, merge.out)
})

in.via(graph).runWith(out)
```

知乎
www.zhihu.com

Akka Stream in Action

知乎
www.zhihu.com

Indexer Architecture

Indexer Architecture

知乎
www.zhihu.com

Binlog listener

Indexer Architecture

知乎
www.zhihu.com

Indexer Architecture

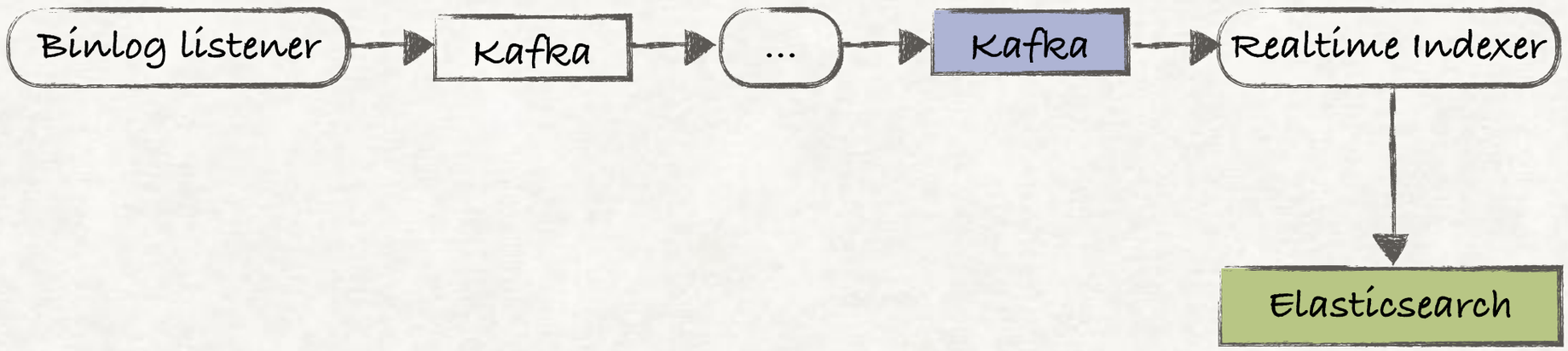
知乎
www.zhihu.com

Indexer Architecture

知乎
www.zhihu.com

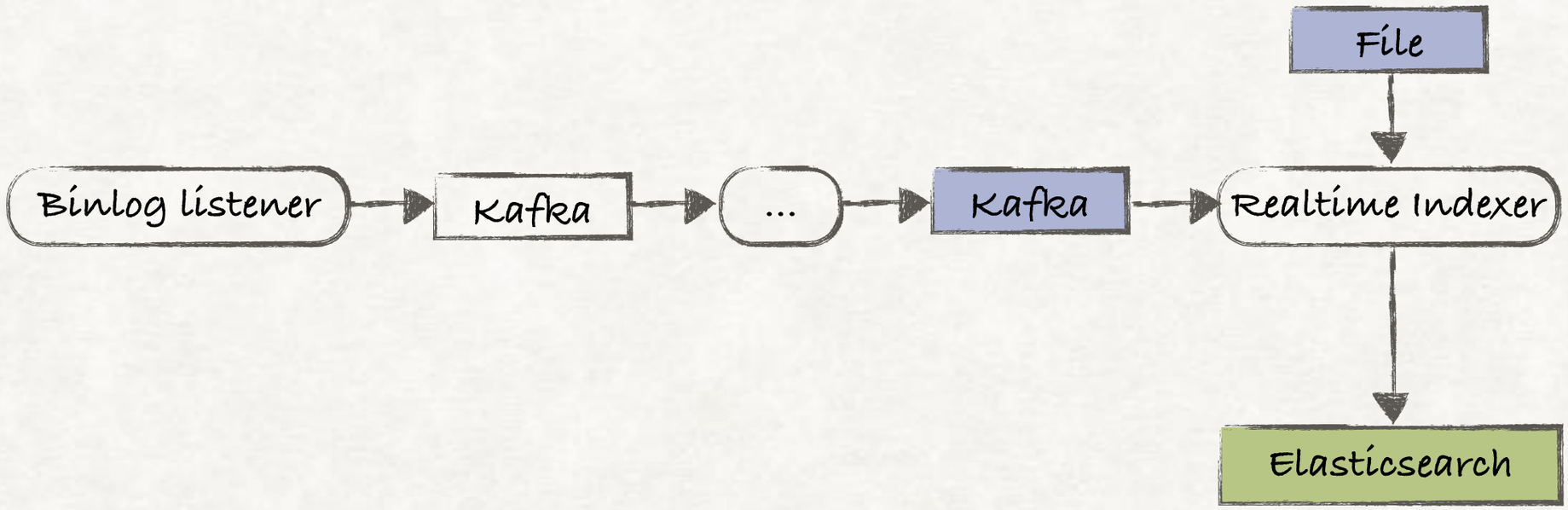
Indexer Architecture

知乎
www.zhihu.com



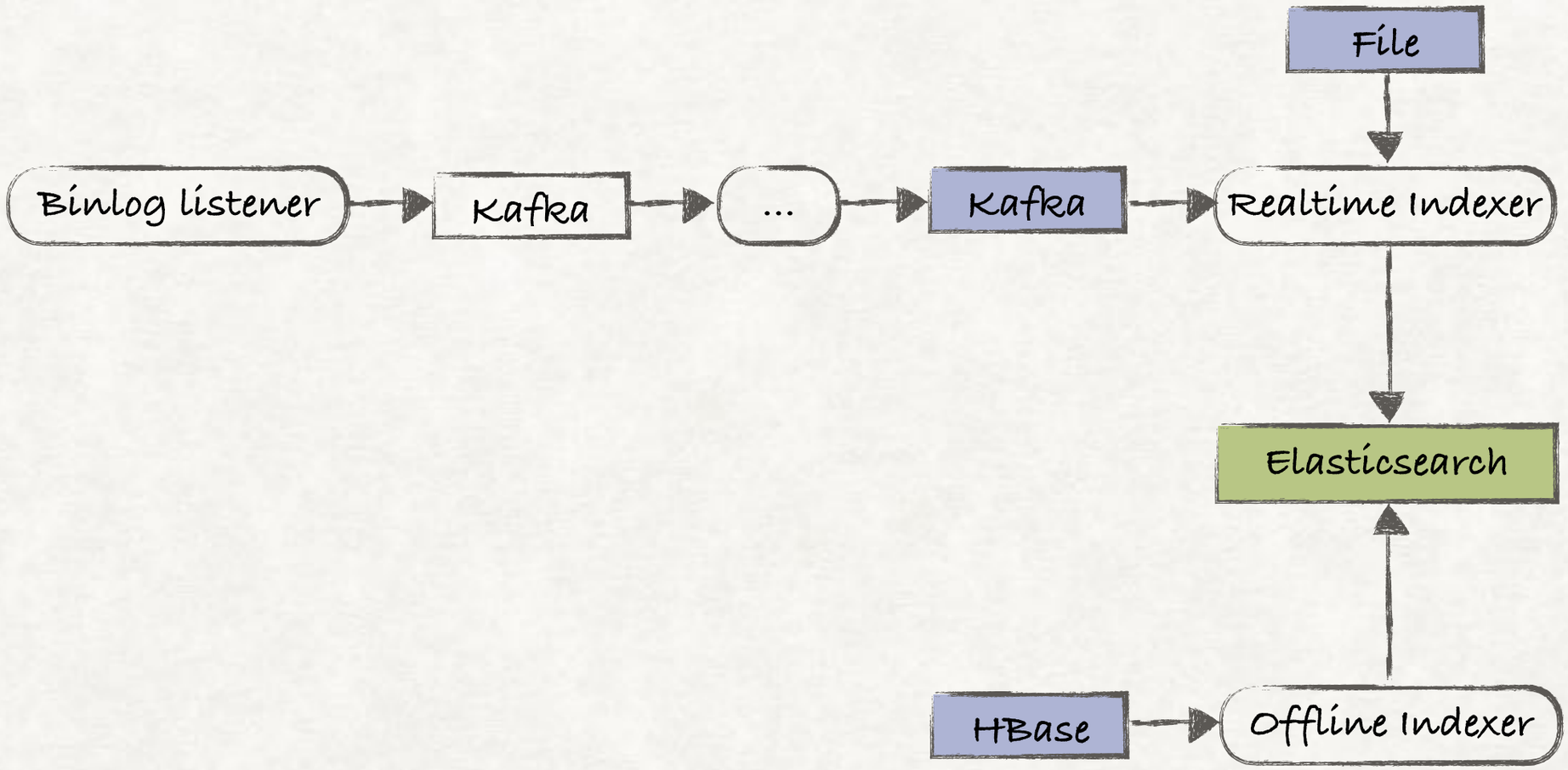
Indexer Architecture

知乎
www.zhihu.com



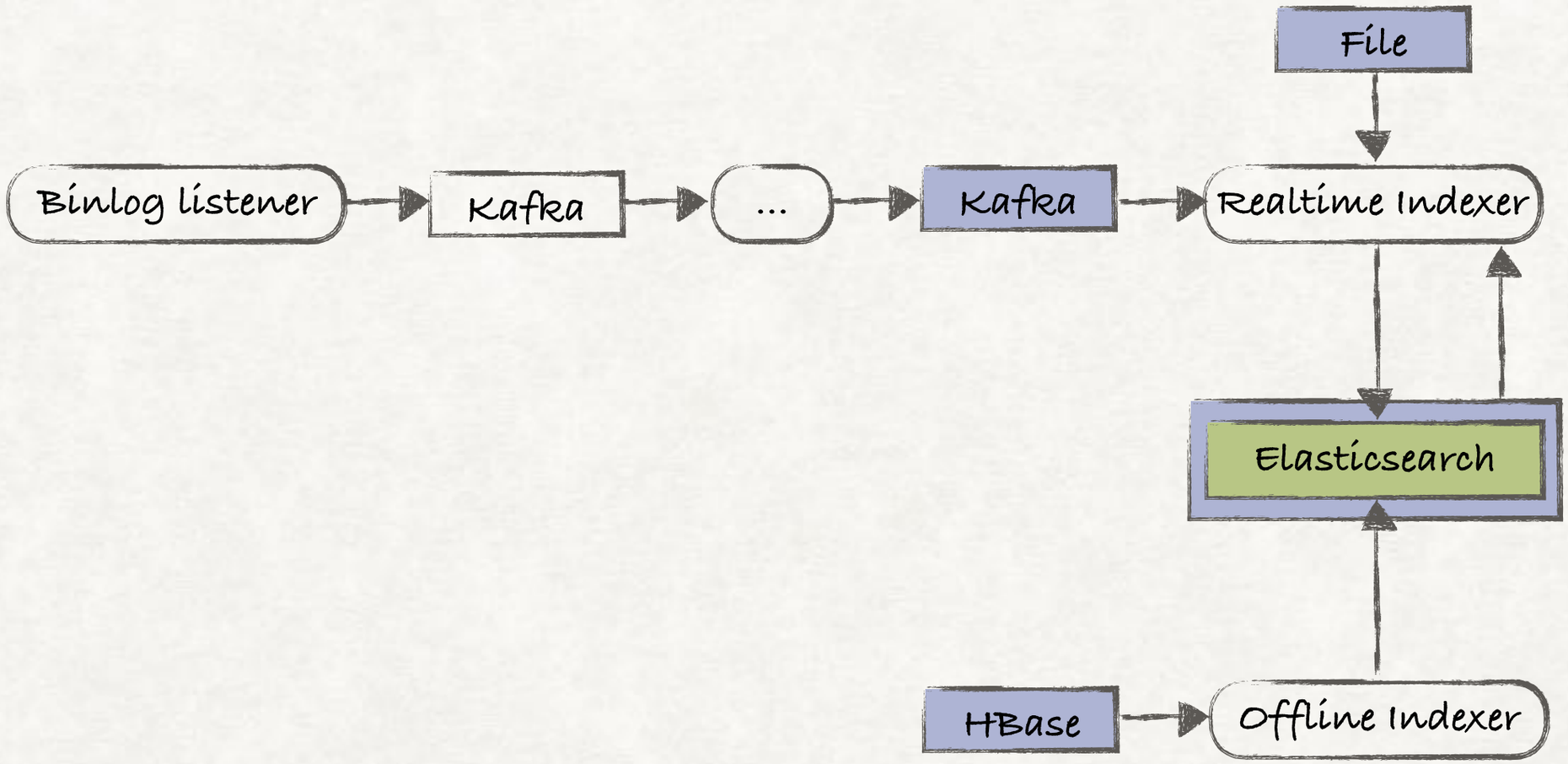
Indexer Architecture

知乎
www.zhihu.com



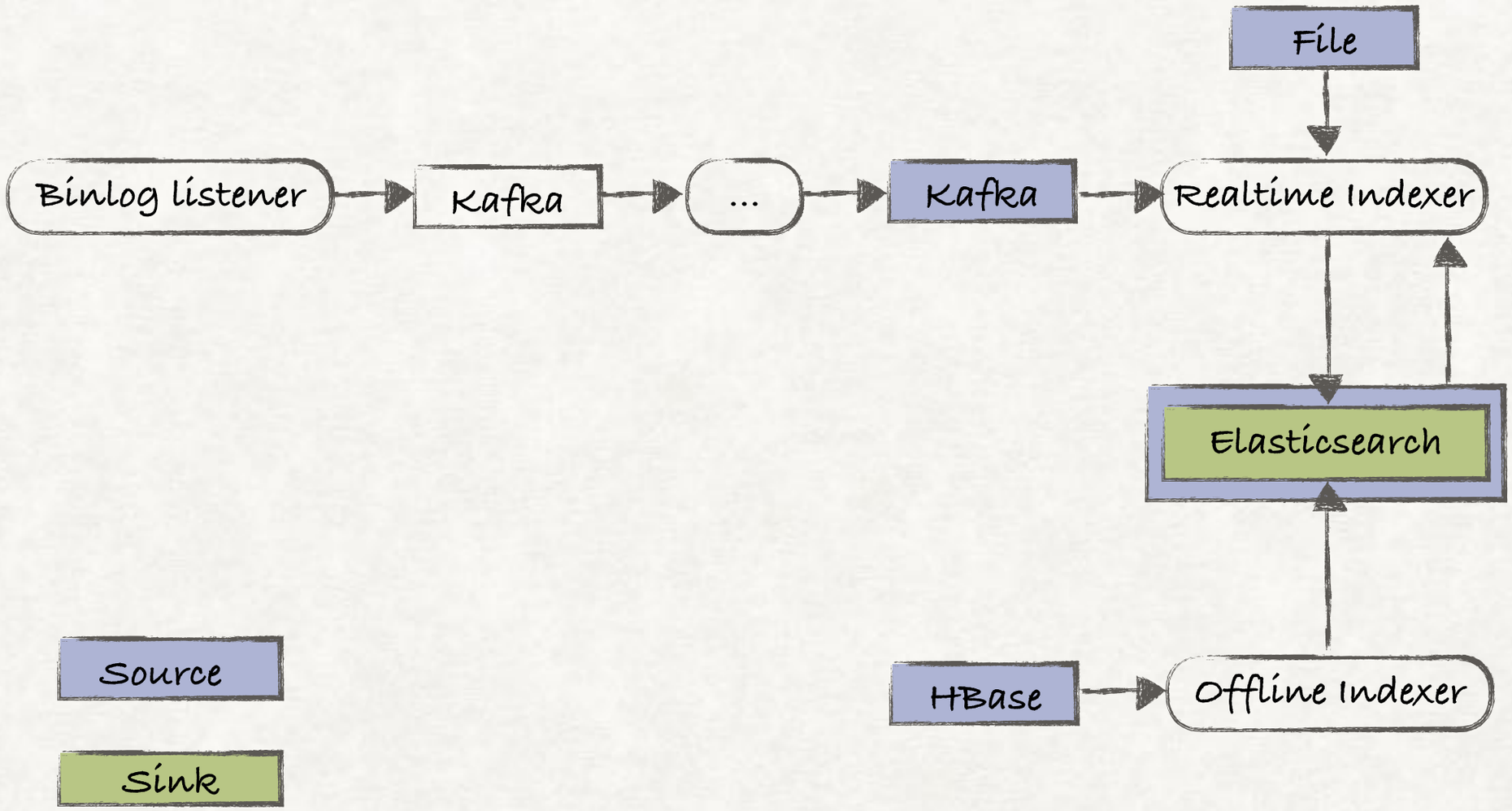
Indexer Architecture

知乎
www.zhihu.com



Indexer Architecture

知乎
www.zhihu.com



Index Sources

知乎
www.zhihu.com

```
def fromLocalFile(path: String ...): Source[SegmentedDoc, _] = {
  val fileDelimiter: Flow[ByteString, ByteString, NotUsed] =
    Framing.delimiter(
        ByteString(System.lineSeparator),
        MAX_FRAME_LENGTH_IN_BYTES,
        true)
  val jsonParserFlow: Flow[ByteString, SegmentedDoc, NotUsed] = ...

  FileIO.fromPath(Paths.get(path))
    .via(fileDelimiter)
    .via(jsonParserFlow)
}
```

Index Sources

知乎
www.zhihu.com

```
def fromLocalFile(path: String ...): Source[SegmentedDoc, _] = {
  val fileDelimiter: Flow[ByteString, ByteString, NotUsed] =
    Framing.delimiter(
        ByteString(System.lineSeparator),
        MAX_FRAME_LENGTH_IN_BYTES,
        true)
  val jsonParserFlow: Flow[ByteString, SegmentedDoc, NotUsed] = ...

  FileIO.fromPath(Paths.get(path))
    .via(fileDelimiter)
    .via(jsonParserFlow)
}

def fromHbase(...): Source[SegmentedDoc, _] = {
  val hbasePublisher = new HbasePublisher(props)
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Source.fromPublisher(hbasePublisher)
    .map(_.bytes)
    .via(pbParserFlow)
}
```

Index Sources

知乎
www.zhihu.com

```
def fromLocalFile(path: String ...): Source[SegmentedDoc, _] = {
  val fileDelimiter: Flow[ByteString, ByteString, NotUsed] =
    Framing.delimiter(
        ByteString(System.lineSeparator),
        MAX_FRAME_LENGTH_IN_BYTES,
        true)
  val jsonParserFlow: Flow[ByteString, SegmentedDoc, NotUsed] = ...

  FileIO.fromPath(Paths.get(path))
    .via(fileDelimiter)
    .via(jsonParserFlow)
}

def fromHbase(...): Source[SegmentedDoc, _] = {
  val hbasePublisher = new HbasePublisher(props)
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Source.fromPublisher(hbasePublisher)
    .map(_.bytes)
    .via(pbParserFlow)
}

def fromKafka(...): Source[SegmentedDoc, _]  = {
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Consumer.committableSource(consumerSettings, Subscriptions.topics(topic))
    .map(_.record.value)
    .via(pbParserFlow)
}
```

Index Sources

知乎
www.zhihu.com

```
def fromLocalFile(path: String ...): Source[SegmentedDoc, _] = {
  val fileDelimiter: Flow[ByteString, ByteString, NotUsed] =
    Framing.delimiter(
        ByteString(System.lineSeparator),
        MAX_FRAME_LENGTH_IN_BYTES,
        true)
  val jsonParserFlow: Flow[ByteString, SegmentedDoc, NotUsed] = ...

  FileIO.fromPath(Paths.get(path))
    .via(fileDelimiter)
    .via(jsonParserFlow)
}

def fromHbase(...): Source[SegmentedDoc, _] = {
  val hbasePublisher = new HbasePublisher(props)
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Source.fromPublisher(hbasePublisher)
    .map(_.bytes)
    .via(pbParserFlow)
}

def fromKafka(...): Source[SegmentedDoc, _]  = {
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Consumer.committableSource(consumerSettings, Subscriptions.topics(topic))
    .map(_.record.value)
    .via(pbParserFlow)
}
```

Index Sources

知乎
www.zhihu.com

```
def fromLocalFile(path: String ...): Source[SegmentedDoc, _] = {
  val fileDelimiter: Flow[ByteString, ByteString, NotUsed] =
    Framing.delimiter(
        ByteString(System.lineSeparator),
        MAX_FRAME_LENGTH_IN_BYTES,
        true)
  val jsonParserFlow: Flow[ByteString, SegmentedDoc, NotUsed] = ...

  FileIO.fromPath(Paths.get(path))
    .via(fileDelimiter)
    .via(balancer(jsonParserFlow, workNum))
}

def fromHbase(...): Source[SegmentedDoc, _] = {
  val hbasePublisher = new HbasePublisher(props)
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Source.fromPublisher(hbasePublisher)
    .map(_.bytes)
    .via(balancer(pbParserFlow, workNum))
}

def fromKafka(...): Source[SegmentedDoc, _]  = {
  val pbParserFlow: Flow[Array[Byte], SegmentedDoc, NotUsed] = ...

  Consumer.committableSource(consumerSettings, Subscriptions.topics(topic))
    .map(_.record.value)
    .via(balancer(pbParserFlow, workNum))
}
```

Concurrent flow

知乎
www.zhihu.com

```scala
def balancer[In, Out](
  worker: Flow[In, Out, Any],
  workerCount: Int
): Flow[In, Out, NotUsed] = {
  Flow.fromGraph(GraphDSL.create() { implicit b =>
    val balancer = b.add(Balance[In](workerCount))
    val merger = b.add(Merge[Out](workerCount))

    (1 to workerCount).foreach { _ =>
      balancer ~> worker.async ~> merge
    }

    FlowShape(balancer.in, merger.out)
  })
}
```

Concurrent flow

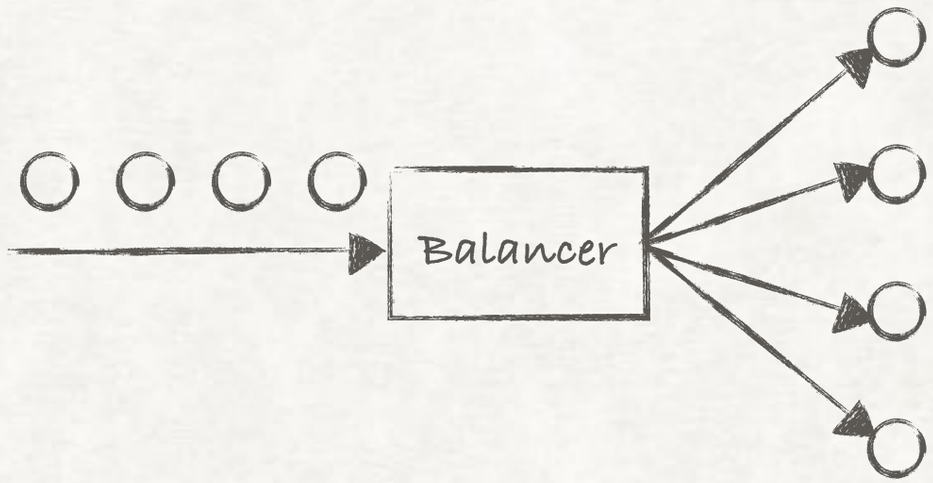
知乎
www.zhihu.com

Concurrent flow

知乎
www.zhihu.com

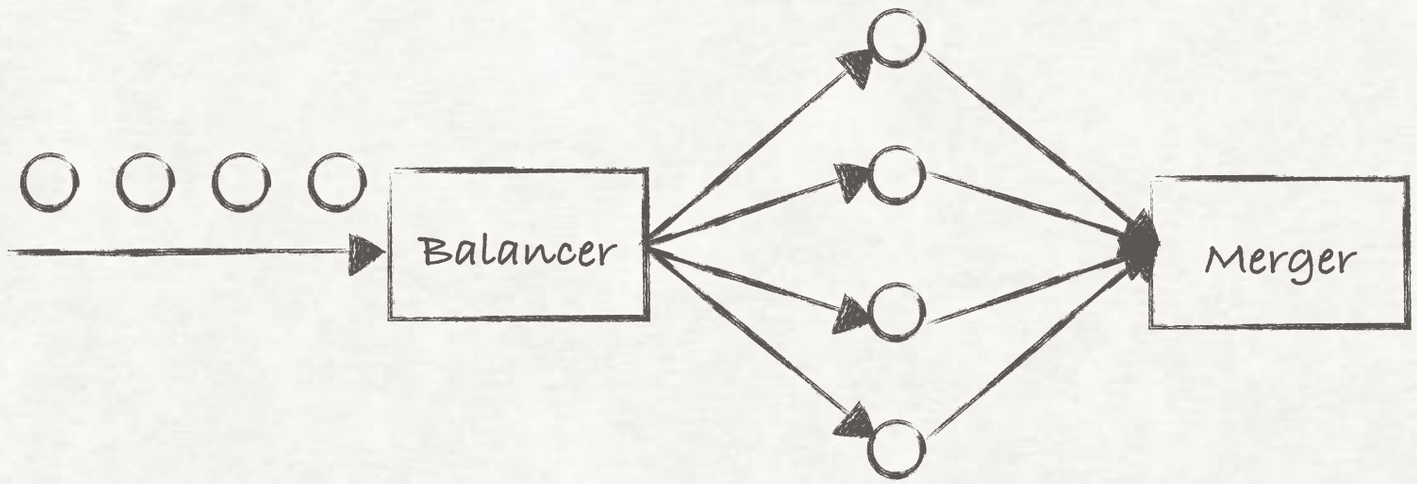
Concurrent flow

知乎
www.zhihu.com



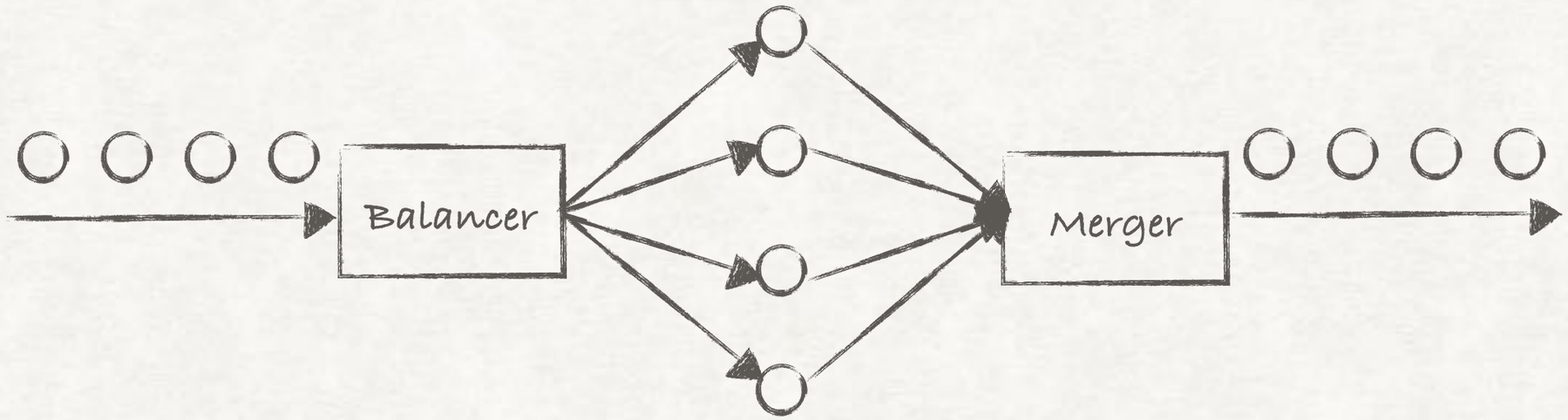
Concurrent flow

知乎
www.zhihu.com



Concurrent flow

知乎
www.zhihu.com



Subscriber (a.k.a Sink)

知乎
www.zhihu.com

```
val source: Source[SegmentedDoc, NotUsed] = IndexSources.create(...)

val subscriber: Subscriber[SegmentedDoc] = ESClient.createSubscriber(...)

source
  .via(balancer(..., workerNum))
  .runWith(Sink.fromSubscriber(subscriber))
```

Demo

知乎
www.zhihu.com

Thanks!