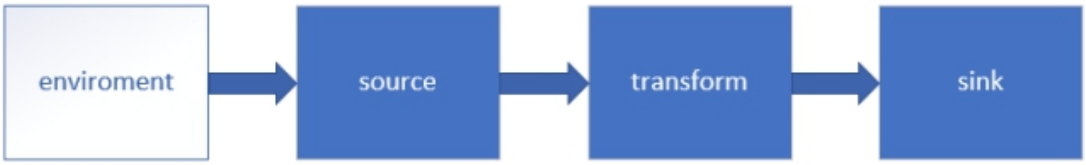
> 人生有两大悲剧，一是万念俱灰，而是踌躇满志

# FlinkAPI之Environment-Source-Transform-Sink




## Environment

### getExecutionEnvironment

> 创建一个执行环境，表示**当前执行程序的上下文**。 如果程序是独立调用的，则此方法返回本地执行环境；如果从命令行客户端调用程序以提交到集群，则此方法返回此集群的执行环境，也就是说，getExecutionEnvironment会根据查询运行的方式决定返回什么样的运行环境，是最常用的一种创建执行环境的方式。

环境直接理解为程序运行所依赖的包和变量的集合

#### 批处理环境

```
val env: ExecutionEnvironment = ExecutionEnvironment.getExecutionEnvironment
```

#### 流处理环境

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
```

### createLocalEnvironment

```
// 返回本地执行环境，需要在调用时指定默认的并行度。
val env = StreamExecutionEnvironment.createLocalEnvironment(1)
```

### createRemoteEnvironment

```
// 返回集群执行环境，将Jar提交到远程服务器。需要在调用时指定JobManager的IP和端口号，并指定要在集群中运行的Jar包。
val env = ExecutionEnvironment
    .createRemoteEnvironment("jobmanage-hostname",
6123,"YOURPATH//wordcount.jar")
```

## Source

## 从集合读取数据

```
// 定义样例类，传感器id，时间戳，温度
case class SensorReading(id: String, timestamp: Long, temperature: Double)

object Sensor {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream1 = env
      .fromCollection(List(
        SensorReading("sensor_1", 1547718199, 35.8),
        SensorReading("sensor_6", 1547718201, 15.4),
        SensorReading("sensor_7", 1547718202, 6.7),
        SensorReading("sensor_10", 1547718205, 38.1)
      ))

    stream1.print("stream1:").setParallelism(1)

    env.execute()
  }
}
```

## 从文件读取数据

```
val stream2 = env.readTextFile("YOUR_FILE_PATH")
```

## 以kafka消息队列的数据作为来源

引入依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.11_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

代码

```
val properties = new Properties()
properties.setProperty("bootstrap.servers", "localhost:9092")
properties.setProperty("group.id", "consumer-group")
properties.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer")
properties.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer")
properties.setProperty("auto.offset.reset", "latest")

val stream3 = env.addSource(
    new FlinkKafkaConsumer011[String]("sensor", new SimpleStringSchema(),
properties))
```

## 自定义Source

> 除了以上的source数据来源，我们还可以自定义source。需要做的，只是传入一个SourceFunction就可以。具体调用如下

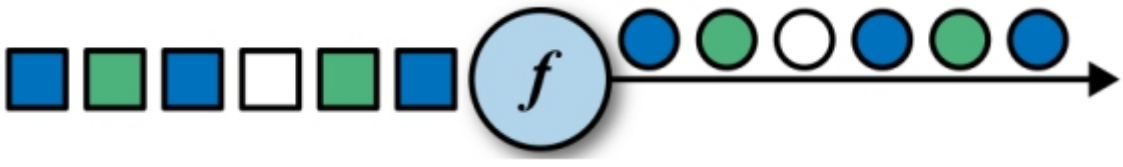
```
val stream4 = env.addSource( new MySensorSource() )
```

需求：定义可以随机生成传感器数据，MySensorSource具体的代码实现如下

```
class MySensorSource extends SourceFunction[SensorReading]{

// flag: 表示数据源是否还在正常运行
var running: Boolean = true

override def cancel(): Unit = {
running = false
}

override def run(ctx: SourceFunction.SourceContext[SensorReading]): Unit = {
// 初始化一个随机数发生器
val rand = new Random()

var curTemp = 1.to(10).map(
i => ( "sensor_" + i, 65 + rand.nextGaussian() * 20 )
)

while(running){
// 更新温度值
curTemp = curTemp.map(
t => (t._1, t._2 + rand.nextGaussian() )
)
// 获取当前时间戳
val curTime = System.currentTimeMillis()

curTemp.foreach(
t => ctx.collect(SensorReading(t._1, curTime, t._2))
)
Thread.sleep(100)
}
}
}
```

# Transform

## map



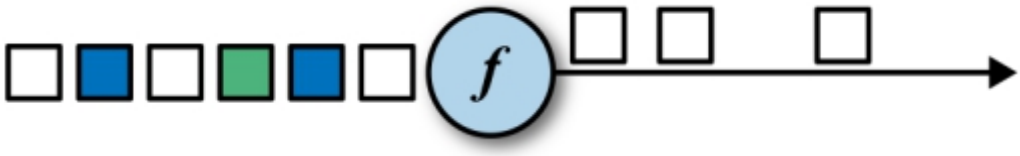

```
val streamMap = stream.map { x => x * 2 }
```

## flatMap

> flatMap的函数签名：def flatMapA,B(f: A ⇒ List[B]): List[B]
>
> 例如: flatMap(List(1,2,3))(i ⇒ List(i,i))
>
> 结果是List(1,1,2,2,3,3),
>
> 而List("a b", "c d").flatMap(line ⇒ line.split(" "))
>
> 结果是List(a, b, c, d)。

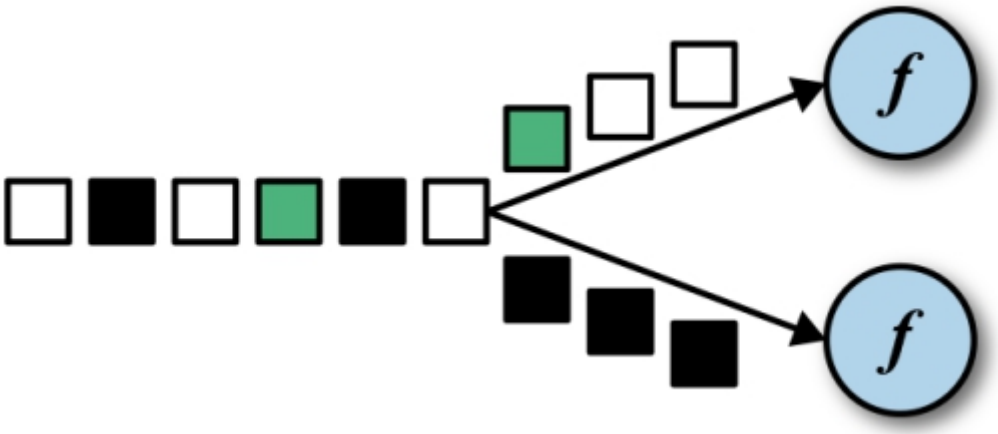
```
val streamFlatMap = stream.flatMap{
    x => x.split(" ")
}
```

## Filter




```
val streamFilter = stream.filter{
    x => x == 1
}
```

## KeyBy




> **DataStream → KeyedStream**：逻辑地将一个流拆分成不相交的分区，每个分区包含具有相同key的元素，在内部以hash的形式实现的。

## 滚动聚合算子（Rolling Aggregation）

这些算子可以针对KeyedStream的每一个支流做聚合。

- sum()
- min()
- max()
- minBy()
- maxBy()

面试题：min和minBy区别

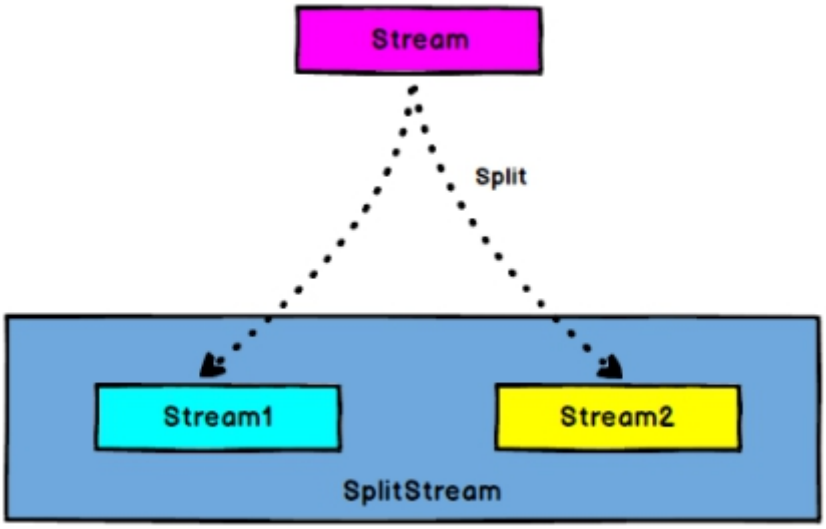
> min():获取的最小值，指定的field是最小，但不是最小的那条记录，后面的示例会清晰的显示。
>
> minBy():获取的最小值，同时也是最小值的那条记录。
>
> 【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

## Reduce

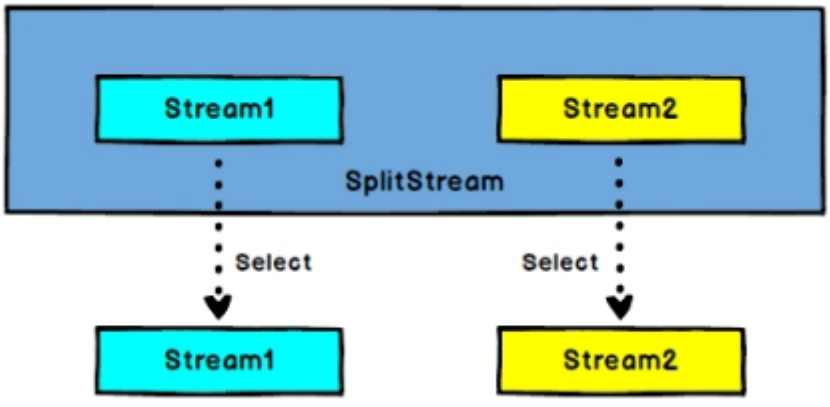
> **KeyedStream → DataStream**：一个分组数据流的聚合操作，合并当前的元素和上次聚合的结果，产生一个新的值，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

```
val stream2 = env.readTextFile("YOUR_PATH\\sensor.txt")
  .map( data => {
    val dataArray = data.split(",")
    SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
  })
  .keyBy("id")
  .reduce( (x, y) => SensorReading(x.id, x.timestamp + 1, y.temperature) )
```

## Split 和 Select




> Split：**DataStream → SplitStream**：根据某些特征把一个DataStream拆分成两个或者多个DataStream。

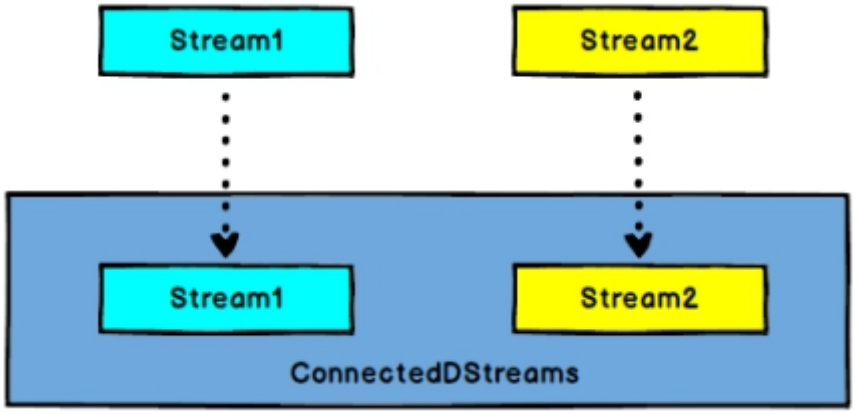



> Select：**SplitStream→DataStream**：从一个SplitStream中获取一个或者多个DataStream。

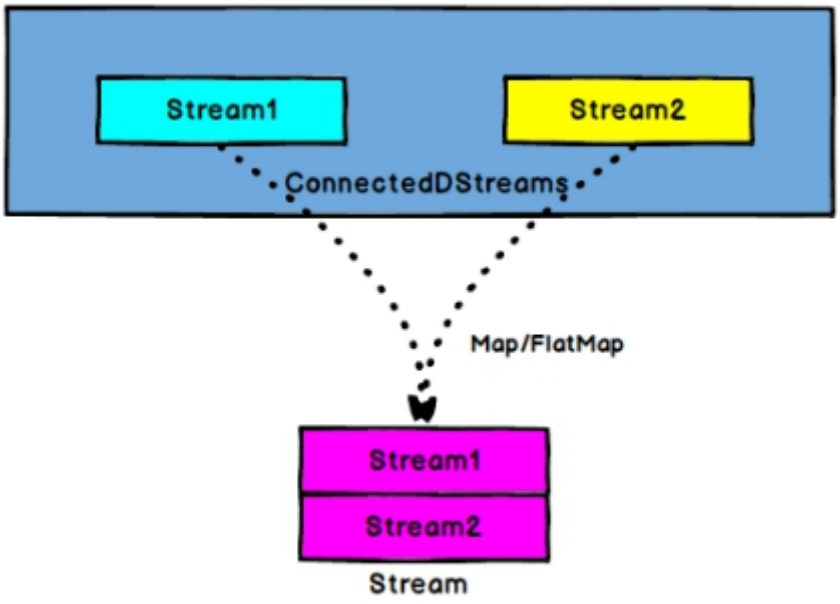
需求：传感器数据按照温度高低（以30度为界），拆分成两个流。

```
val splitStream = stream2
  .split( sensorData => {
    if (sensorData.temperature > 30) Seq("high") else Seq("low")
  } )

val high = splitStream.select("high")
val low = splitStream.select("low")
val all = splitStream.select("high", "low")
```

## Connect和CoMap



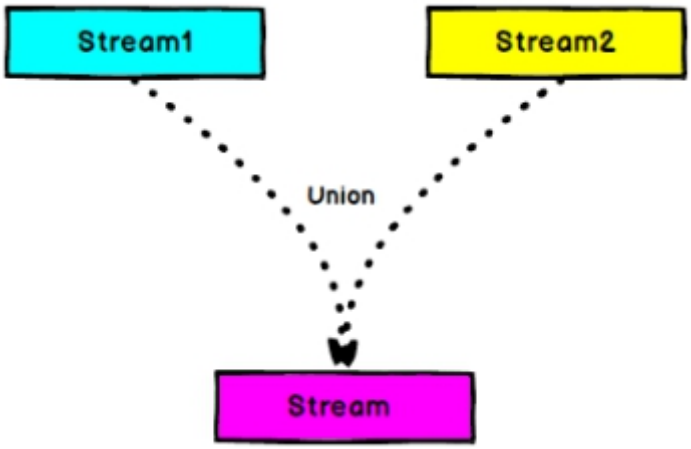

> Connect：**DataStream,DataStream → ConnectedStreams**：连接两个保持他们类型的数据流，两个数据流被Connect之后，只是被放在了一个同一个流中，内部依然保持各自的数据和形式不发生任何变化，两个流相互独立。




> CoMap：**ConnectedStreams → DataStream**：作用于ConnectedStreams上，功能与map和flatMap一样，对ConnectedStreams中的每一个Stream分别进行map和flatMap处理。

```
val warning = high.map( sensorData => (sensorData.id, sensorData.temperature) )
val connected = warning.connect(low)

val coMap = connected.map(
    warningData => (warningData._1, warningData._2, "warning"),
    lowData => (lowData.id, "healthy")
)
```

## Union




> **DataStream → DataStream**：对两个或者两个以上的DataStream进行union操作，产生一个包含所有DataStream元素的新DataStream。

```
//合并以后打印
val unionStream: DataStream[StartUpLog] = appStoreStream.union(otherStream)
unionStream.print("union:::")
```

**Connect与Union 区别：**

1.Union之前两个流的类型必须是一样，Connect可以不一样，在之后的coMap中再去调整成为一样的。

2.Connect只能操作两个流，Union可以操作多个。

# Sink

> **Flink没有类似于spark中foreach方法**，让用户进行迭代的操作。虽有对外的输出操作都要利用Sink完成。最后通过类似如下方式完成整个任务最终输出操作。

```
stream.addSink(new MySink(xxxx))
```

官方提供了一部分的框架的sink。除此以外，需要用户自定义实现sink！！！

## Bundled Connectors

Connectors provide code for interfacing with various third-party systems. Currently these systems are supported:

- Apache Kafka (source/sink)
- Apache Cassandra (sink)
- Amazon Kinesis Streams (source/sink)
- Elasticsearch (sink)
- Hadoop FileSystem (sink)
- RabbitMQ (source/sink)
- Apache NiFi (source/sink)
- Twitter Streaming API (source)

## Connectors in Apache Bahir

Additional streaming connectors for Flink are being released through Apache Bahir, including:

- Apache ActiveMQ (source/sink)
- Apache Flume (sink)
- Redis (sink)
- Akka (sink)
- Netty (source)

### KafkaSink

pom.xml

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.11_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

```
val union = high.union(low).map(_.temperature.toString)

union.addSink(new FlinkKafkaProducer011[String]("localhost:9092", "test", new
SimpleStringSchema()))
```

### RedisSink

pom.xml

```
<dependency>
    <groupId>org.apache.bahir</groupId>
    <artifactId>flink-connector-redis_2.11</artifactId>
    <version>1.0</version>
</dependency>
```

定义一个redis的mapper类，用于定义保存到redis时调用的命令

```
class MyRedisMapper extends RedisMapper[SensorReading]{
  override def getCommandDescription: RedisCommandDescription = {
    new RedisCommandDescription(RedisCommand.HSET, "sensor_temperature")
  }
  override def getValueFromData(t: SensorReading): String =
t.temperature.toString

  override def getKeyFromData(t: SensorReading): String = t.id
}
```

主函数

```
val conf = new
FlinkJedisPoolConfig.Builder().setHost("localhost").setPort(6379).build()
dataStream.addSink( new RedisSink[SensorReading](conf, new MyRedisMapper) )
```

## ElasticsearchSink

pom.xml

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-elasticsearch6_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

```
al httpHosts = new util.ArrayList[HttpHost]()
httpHosts.add(new HttpHost("localhost", 9200))

val esSinkBuilder = new ElasticsearchSink.Builder[SensorReading]( httpHosts, new
ElasticsearchSinkFunction[SensorReading] {
  override def process(t: SensorReading, runtimeContext: RuntimeContext,
requestIndexer: RequestIndexer): Unit = {
    println("saving data: " + t)
    val json = new util.HashMap[String, String]()
    json.put("data", t.toString)
    val indexRequest =
Requests.indexRequest().index("sensor").`type`("readingData").source(json)
    requestIndexer.add(indexRequest)
    println("saved successfully")
  }
} )
dataStream.addSink( esSinkBuilder.build() )
```

## 自定义Sink连接JDBC【重点掌握】

pom.xml

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.44</version>
</dependency>
```

添加MyJdbcSink

```
class MyJdbcSink() extends RichSinkFunction[SensorReading]{
  var conn: Connection = _
  var insertStmt: PreparedStatement = _
  var updateStmt: PreparedStatement = _

  // open 主要是创建连接
  override def open(parameters: Configuration): Unit = {
    super.open(parameters)

    conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test",
"root", "123456")
    insertStmt = conn.prepareStatement("INSERT INTO temperatures (sensor, temp)
VALUES (?, ?)")
    updateStmt = conn.prepareStatement("UPDATE temperatures SET temp = ? WHERE
sensor = ?")
  }
  // 调用连接，执行sql
  override def invoke(value: SensorReading, context: SinkFunction.Context[_]):
Unit = {

updateStmt.setDouble(1, value.temperature)
    updateStmt.setString(2, value.id)
    updateStmt.execute()

    if (updateStmt.getUpdateCount == 0) {
      insertStmt.setString(1, value.id)
      insertStmt.setDouble(2, value.temperature)
      insertStmt.execute()
    }
  }

  override def close(): Unit = {
    insertStmt.close()
    updateStmt.close()
    conn.close()
  }
}
```

主程序

```
dataStream.addSink(new MyJdbcSink())
```