

Chapter 5 The DataStream API (v1.7)

本章前言：

从以下几个方面介绍 flink 的使用

*DataStream API

*构建典型的 flink 流应用

*数据展示和分区转换

*窗口算子

*基于时间的转换

*状态算子

目标：使用基础算子实现一个流应用

一、 Hello, Flink!

构建典型的 flink 应用程序的 5 个步骤

(以每 5s 求传感器平均值为例)

1.配置执行环境

2.读取一个或多个数据源

3.使用流转换来实现应用逻辑

4.可选择的将结果输出到一个或者多个 sink

5.执行代码

下面分别展开来说

1. Set Up the Execution Environment 配置执行环境

a、execution environment 决定了程序是在本地还是一个集群中

b、本地与远程执行环境的配置

* 第一种方式：在流 API 中执行环境的用 StreamExecutionEnvironment 类来表示,它的 getExecutionEnvironment()方法可以用来检索执行环境，此方法可以返回一个本地或者远程的执行环境，具体取决于调用该方法时的 context,例如：如果是与集群远程连接的 client 调用了该方法则返回一个远程客户端，相反它返回一个本地的环境,代码如下所示

```
// set up the streaming execution environment  
val env = StreamExecutionEnvironment.getExecutionEnvironment
```

* 第二种方式：显示声明的方式，如图

创建本地和远程环境的示例

```
// create a local stream execution environment
val localEnv:
  StreamExecutionEnvironment.createLocalEnvironment()

// create a remote stream execution environment
val remoteEnv =
  StreamExecutionEnvironment.createRemoteEnvironment(
    "host",           // hostname of JobManager
    1234,             // port of JobManager
    process(
      "path/to/jarFile.jar") // JAR file to ship to
    the JobManager
```

c 、设置时间语义 : env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

2.Read an Input Stream 读取输入流

*env.addSource(new SensorSource)添加一个样例类数据源(只要是远远不断的产生数据), flink 支持的数据有很多种, 将在后面讨论

*env.assignTimestampsAndWatermarks(new SensorTimeAssigner)分配时间戳和水印, 暂不讨论 SensorTimeAssigner 的细节

3.Apply Transformations (使用转换)

*transformations 可以产生新的流, 可以把 records 分区, 应用程序的逻辑通过 transformations 链来定义, 示例中的 transformations 体现如下

map(将温度转换为摄氏度)——>keyBy(按照传感器的 ID 进行分区)——>timeWindow(将每个分区的数据划分为 5s 的翻转窗口)

*示例中使用了自定义函数来计算每个窗口的平均温度

4. Output the Result (输出结果)

*可以写入多种不同的 sink 中，也可以自己实现 sink

*sink 的选择和 checkpoint 算法的配合会影响程序的端到端一致性

5. Execute

*flink 是懒惰执行的，创建流源和转换的 API 调用不会立即触发任何数据处理，相反，API 的调用是为了在环境中构造执行计划，该计划包含了 source 和所有的 transformations，只有在 execute()方法被调用时才会真正触发系统执行逻辑程序

*被构造的计划叫做 JobGraph，它被提交给 JobManager 执行，JM 根据环境不同会在本地启动线程或者把 JobGraph 发送给远程的 JM，如果是远程的需要 JobGraph 和 Jar 包需要一起提供

例子整体代码

cation ingests a stream of temperature measurements from multiple sensors.

First, let's have a look at the data type we will be using to represent sensor readings:

```
case class SensorReading(添加数据源  
    根据事件事件  
    分配时间戳和水印  
    id: String,  
    timestamp: Long,  
    temperature: Double)
```

The program in [Example 5-1](#) converts the temperatures from Fahrenheit to Celsius and computes the average temperature every 5 seconds for each sensor.
5-1 把传感器从华氏度转为摄氏度 并每5s计算平均值

Example 5-1. Compute the average temperature every 5 seconds for a stream of sensors

```
Transformations  
// Scala object that defines the DataStream  
program in the main() method.  
object AverageSensorReadings {  
  
    // main() defines and executes the DataStream  
    program  
    def main(args: Array[String]) {  
  
        // set up the streaming execution 程序真正的执行  
        environment  
        val env =  
        StreamExecutionEnvironment.getExecutionEnvir  
        onment  
        // use event time for the application  
  
        env.setStreamTimeCharacteristic(TimeCharacter  
        istic.EventTime)  
  
        // create a DataStream[SensorReading] from a  
        stream source  
        val sensorData: DataStream[SensorReading]  
        = env  
            // ingest sensor readings with a  
            SensorSource SourceFunction  
            .addSource(new SensorSource)  
            // assign timestamps and watermarks  
            (required for event time)  
            .assignTimestampsAndWatermarks(new  
            SensorTimeAssigner)  
  
        val avgTemp: DataStream[SensorReading] =  
        sensorData  
            // convert Fahrenheit to Celsius with an  
            inline lambda function  
            .map( r => {  
                val celsius = (r.temperature - 32) *  
                (5.0 / 9.0)  
                SensorReading(r.id, r.timestamp,  
                celsius)  
            } )  
            // organize readings by sensor id  
            .keyBy(_.id)  
            // group readings in 5 second tumbling  
            windows  
            .timeWindow(Time.seconds(5))  
            // compute average temperature using a  
            user-defined function  
            .apply(new TemperatureAverager)  
            // print result stream to standard out  
            avgTemp.print()  
            // execute application  
            env.execute("Compute average sensor  
            temperature")  
    }  
}
```

二、 Transformations

前言：

*编写 Data Streaming API 本质上是使用 transformation 创造 dataflow graph 来实现业务逻辑

*大多数的 stream transformations 是基于用户自定义的函数，接口一般被定义为 SAM (single abstract method)

方式，java 8 中可以使用 lambda 实现，也支持 scala lambda 函数，还有一种方式是传入自己实现的函数类对象

(例如：readings.map(new MyMapFunction)， MyMapFunction 就是对 MapFunction 类的实现)

*transformations DataStream API 在以下四个方面来介绍

(1)Basic transformations 是单个 record 的的转换

(2)KeyedStream 转换是应用在 key 的 context event 的转换

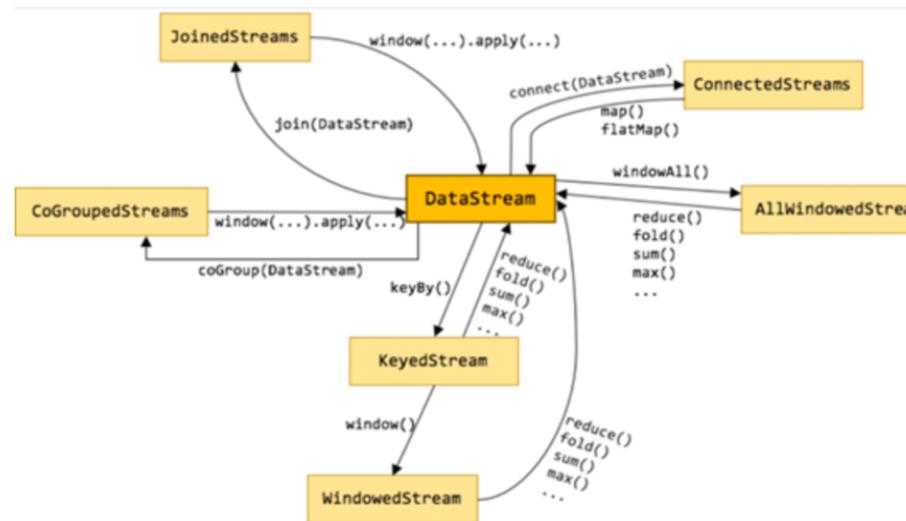
(3)Multistream transformations 将多个流合并为一个流，或将一个流拆分为多个流

(4) Distribution transformations 重组了流事件

补充

1. DataStream Transformation

1.1 DataStream转换关系



(1) Basic Transformations

*Map : DataStream.map()返回一个新的 DataStream, 对单条 record 进行操作

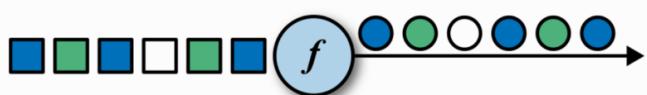


Figure 5-1. A map operation that transforms every square into a circle of the same color

map 函数效果图

MapFuncation 接口

```
MapFunction接口  
// T: the type of input elements 输入与输出的类型  
// O: the type of output elements  
MapFunction[T, O]  
> map(T): O 接口内部定义的map函数
```

使用方式一：获取 record 的 id

```
val readings: DataStream[SensorReading] = ...  
val sensorIds: DataStream[String] = readings.map(new  
MyMapFunction)  
class MyMapFunction extends  
MapFunction[SensorReading, String] {  
    override def map(r: SensorReading): String = r.id  
}
```

使用方式一：实现MapFuncation接口

使用方式二：获取 record 的 id

```
val readings: DataStream[SensorReading] = ...  
val sensorIds: DataStream[String] =  
readings.map(r => r.id) 直接使用lambda表达式方式
```

*Filter : DataStream.filter () 返回满足条件的 records

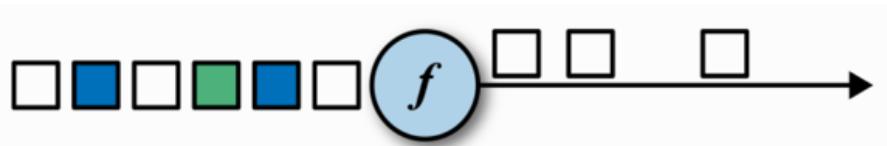


Figure 5-2. A filter operation that only retains white values

filter 示意图

```
// T: the type of elements
FilterFunction[T]
  > filter(T): Boolean
```

Filter 接口

```
val readings: DataStream[SensorReadings] = ...
val filteredSensors = readings
  .filter( r => r.temperature >= 25 )
```

使用方式

*FlatMap：可以对进入的一条 record 产生 0、1 或者多条返回 record，它是 map 和 filter 的概括可以满足这两个操作

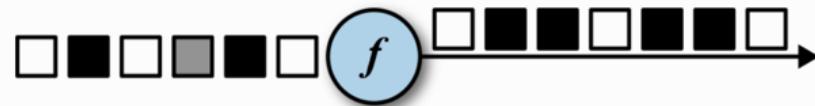


Figure 5-3. A flatMap operation that outputs white squares, duplicates black squares, and drops gray squares

flatmap 示意图：白色方块不处理，灰色过滤，黑色复制

basic tf flatmap配图
`// T: the type of input elements // O: the type of output elements`
`FlatMapFunction[T, O]`
`> flatMap(T, Collector[O]): Unit`

FlatMap 接口

```
val sentences: DataStream[String] = ...
val words: DataStream[String] = sentences
  .flatMap(id => id.split(" "))
```

使用方式

2.KeyedStream Transformations 注：可以有状态

*根据 record 的某一特性 (key) 在逻辑上分区为不相交的子流，相同的 key 的子流中的 records 可以访问相同的状态，需防止 key 无限增加即逻辑分区无限增加

(1)keyBy 算子

*会根据 key 进行分区，以便后续的 operator 可以在相同的 task 中处理 records

*keyby 算子内的 record 会根据 key 限定访问的状态范围，即只能访问当前处理的 record 的状态

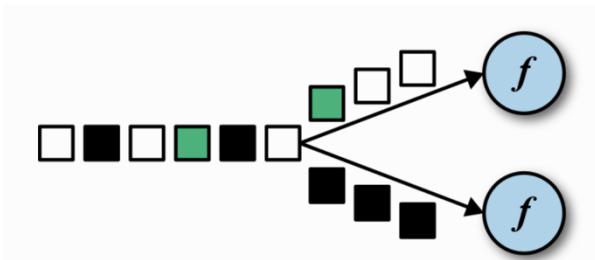


Figure 5-4. A keyBy operation that partitions events based on color

keyBy : 将黑色单独分区其他颜色都进入一个分区

```
val readings: DataStream[SensorReading] = ...
val keyed: KeyedStream[SensorReading, String] =
  readings
    .keyBy(r => r.id)
```

使用方式

(2)Rolling aggregations

*应用在 KeyedStream 上产生一个聚合的 DataStream

*提供 sum()、min()、max()、minBy()、maxBy()返回到目前为止观察到的最小/大值的事件

maxBy()返回到目前为止观察到的最大值的事件

```
val inputStream: DataStream[(Int, Int, Int)] =  
env.fromElements(  
  (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))  
  
val resultStream: DataStream[(Int, Int, Int)] =  
inputStream  
  .keyBy(0) // key on first field of the tuple  
  .sum(1) // sum the second field of the tuple in  
  place  
  key 1 输出 (1-key,7-sum,2-无定义)  
  key 2 输出 (2,5,1), 含义和key1相同
```

Rolling agg 举例

注：(1) 不能组合多个 rolling 聚合方法，一次只能计算一个滚动聚合

(2) only apply a rolling aggregations operator on a stream with a bounded key domain, rolling 聚合只能使用在有限的 key streaming 域中

(3) Reduce 是针对前后两条记录的函数

*reduce tf 是 rolling 聚合的概括，应用在 KeyedStreaming 上

*将前后两条 record 缩减组合在一起并生成一个 DataStream

*reduce 不会改变 stream 的类型

注：(1) only apply a rolling aggregations operator on a stream with a bounded key domain, rolling 聚合只能使用在有限的 key streaming 域中

```
// T: the element type  
ReduceFunction[T]  
    > reduce(T, T): T
```

reduce 接口

```
val inputStream: DataStream[(String,  
List[String])] = env.fromElements(  
    ("en", List("tea")), ("fr", List("vin")), ("en",  
    List("cake")))  
  
val resultStream: DataStream[(String,  
List[String])] = inputStream  
    .keyBy(0) 按照key分区  
    .reduce((x, y) => (x._1, x._2 :: y._2))
```

reduce 举例 将key放在第一位置，前后两条记录的list连接为一个

3. Multistream Transformations 本小节讨论 多输入输出流 的 tf

(1)Union

*可以提取一个或者多个类型相同的流，DataStream.union(Stream1, Stream2), Stream1、Stream2、返回的 Stream 都是相同类型

*事件合并时采用 FIFO 顺序策略，不提供去重

union 示意图 2-1

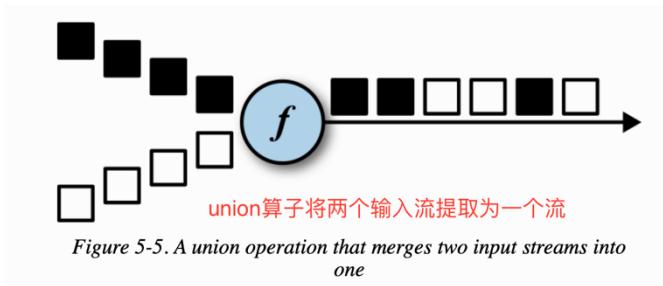


Figure 5-5. A union operation that merges two input streams into one

union 代码使用示意图

```
val parisStream: DataStream[SensorReading] = ...
val tokyoStream: DataStream[SensorReading] = ...
val rioStream: DataStream[SensorReading] = ...
val allCities: DataStream[SensorReading] =
    parisStream
        .union(tokyoStream, rioStream)
```

union代码示意

(2) Connect, coMap, and coFlatMap

前言：举例：森林火灾监控，温度达到设定值且烟雾浓度达到设定值，可以使用 connect 方法通过两个输入流创建一个 ConnectedStreams 对象

a、ConnectStreams 原理

ConnectStreams 对象提供两个接口 CoMapFunction 和 CoFlatMapFunction，这两个接口在处理两个流输入流时

函数是分开的， map1() and flatMap1()处理第一个流， map2() and flatMap2()处理第二个流， 接口样式可看图

```
// IN1: the type of the first input stream
// IN2: the type of the second input stream
// OUT: the type of the output elements
CoMapFunction[IN1, IN2, OUT]
  > map1(IN1): OUT
  > map2(IN2): OUT

// IN1: the type of the first input stream
// IN2: the type of the second input stream
// OUT: the type of the output elements
CoFlatMapFunction[IN1, IN2, OUT]
  > flatMap1(IN1, Collector[OUT]): Unit
  > flatMap2(IN2, Collector[OUT]): Unit

// first stream
val first: DataStream[Int] = ...
// second stream
val second: DataStream[String] = ...

// connect streams
val connected: ConnectedStreams[Int, String] =
  first.connect(second)
```

注：无法控制调用 CoMapFunction 或 CoFlatMapFunction 方法的顺序

b、 ConnectedStream 需与其他算子的配合使用

ConnectedStream 不会对两个流建立某种联系，所以处理后是随意分发到后续的 operator instance，为了达到确定的事件分发路由，connect()需要组合 keyBy()或者 broadcast()

(1) 与 keyBy 配合使用：如图

```

val one: DataStream[(Int, Long)] = ...
val two: DataStream[(Int, String)] = ...

// keyBy two connected streams
val keyedConnect1: ConnectedStreams[(Int, Long),
(Int, String)] = one
  .connect(two)
  .keyBy(0, 0) // key both input streams on first
attribute

// alternative: connect two keyed streams
val keyedConnect2: ConnectedStreams[(Int, Long),
(Int, String)] = one.keyBy(0)
  .connect(two.keyBy(0))

```

使用 keyBy 时需要注意两个 stream 的 key 需要相同的类型，类似与 sql 的 join 语义

(2) 与 broadcast 配合使用：如图：

```

val first: DataStream[(Int, Long)] = ...
val second: DataStream[(Int, String)] = ...

// connect streams with broadcast
val keyedConnect: ConnectedStreams[(Int, Long),
(Int, String)] = first
  // broadcast second input stream
  .connect(second.broadcast()) second流被复制广播

```

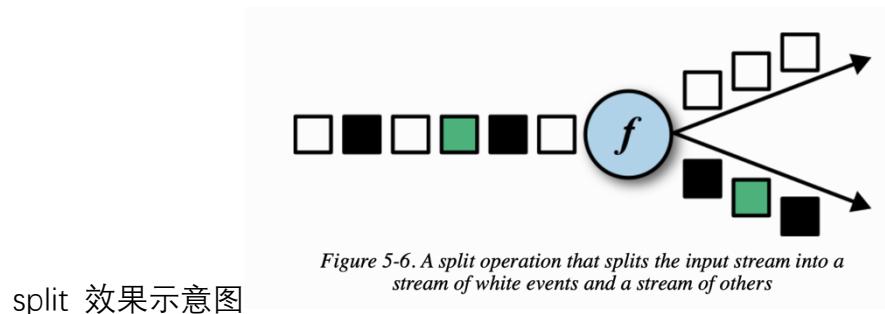
广播流的 events 将被复制广播到所有下游 operator 并行实例，非广播流只是被转发，所以两个流可以被一起处理

注：Broadcast state 是 broadcast()-connect() tf 的改进版本，可以用来更新 rulestream 的规则数据

(3)Split and select

*split 流可以将一个流拆分为多个流，可以是过滤和复制输入流的事件，代码形式如下

*输入和输出的类型一致



split 效果示意图

split方法接收一个OutputSelector，它的select方法返回的 Iterable[String],String指定了这条record的输出路由

```
// IN: the type of the split elements  
OutputSelector[IN]  
> select(IN): Iterable[String]
```

split 接收的 OutputSelect 接口

```
val inputStream: DataStream[(Int, String)] = ...  
  
val splitted: SplitStream[(Int, String)] =  
  inputStream  
    .split(t => if (t._1 > 1000) Seq("large") else  
      Seq("small"))  
  
val large: DataStream[(Int, String)] =  
  splitted.select("large")  
val small: DataStream[(Int, String)] =  
  splitted.select("small")  
val all: DataStream[(Int, String)] =  
  splitted.select("small", "large")
```

split 代码使用示意

4.Distribution Transformations (分布式 tf)

*使用 DataStream API 构建应用程序时，系统会根据操作语义和配置的并行性自动选择数据分区策略，并将数据路由到正确的目的地，但在数据倾斜及一些场景中需要自定义分区策略，本节将介绍自定义分区策略

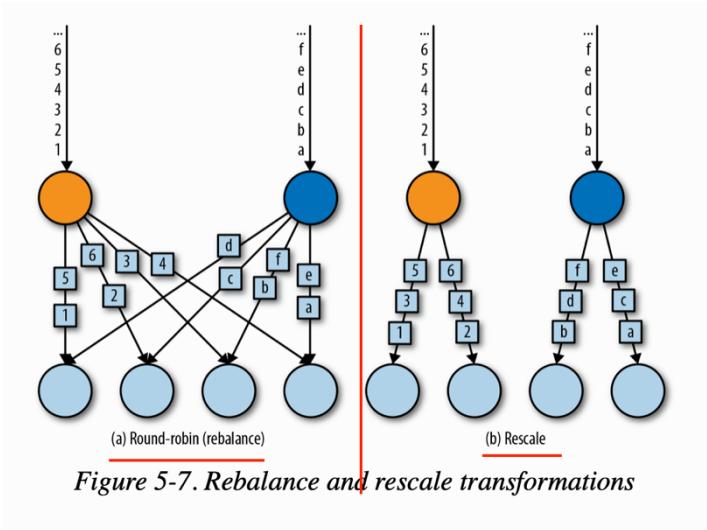
*本节讨论的所有的 tf 都返回 DataStream，keyBy()返回的是 KeyedStream

4.1 Random 方式

由 DataStream.shuffle()方法实现，该方式均匀的分布数据给下游的并行 task

4.2 Round-Robin 方式

轮流的将数据分给并行的 task, 一条数据换一次



4.3 Rescale

将数据发送到下游的部分 task 并不是所有, 如右边 4.3 图中示例

4.4 Broadcast

将数据 copy 所有的输入数据分给全部下游并行的 task

4.5 Global

将输入数据全部发给下游 operator 第一个并行的 task, 因为所有数据都被路由到一个 task, 故会影响性能

所以需谨慎使用

4.6 Custom

以上策略都不合适，可以自定义分区策略

```
val numbers: DataStream[(Int)] = ...
numbers.partitionCustom(myPartitioner, 0)

object myPartitioner extends
Partitioner[Int] {
    val r = scala.util.Random

    override def partition(key: Int,
    numPartitions: Int): Int = {
        if (key < 0) 0 else
        r.nextInt(numPartitions)
    }
}
```

custom 使用示例

三、Setting the Parallelism

设置并行度

前言：

flink 程序是并行的在分布式环境中运行，流程如下

程序提交——>Jobmanager——>系统为算子的执行创建 dataflow graph 并做好准备——>每一个 operator 被并行化为多个 task——>task 的数量叫做 operator 的并行度，它决定了 operator 并行处理能力

a、operator 的并行度和环境并行度

operator 的并行度默认和环境的并行度保持一致也可以单独设置，当环境为 local 时，所有 operator 的并行度和 cpu 内核数一致，当为 flink 集群模式时默认为集群环境的并行度

b、覆盖环境默认的并行度

```
val env: StreamExecutionEnvironment.getExecutionEnvironment  
// set parallelism of the environment  
env.setParallelism(32)
```

c、指定算子进行不同的并行度

访问程序的默认并行度

```
val env: 查询环境的并行度
StreamExecutionEnvironment.getExecutionEnvironment
// get default parallelism as configured in the cluster
config or
// explicitly specified via the submission client.
val defaultP = env.getParallelism
```

指定算子不同的并行度

```
val env =
StreamExecutionEnvironment.getExecutionEnvironment

// get default parallelism 不同算子不同并行度
val defaultP = env.getParallelism

// the source runs with the default parallelism
val result: = env.addSource(new CustomSource)
// the map parallelism is set to double the default
parallelism
.map(new MyMapper).setParallelism(defaultP * 2)
// the print sink parallelism is fixed to 2
.print().setParallelism(2)
```

四、Types

可参考数据类型与序列化（以整理在 flink 文件夹）

前言：

*flink 处理的 dataStream 被定义为事件对象的数据流，为了处理这些对象需要序列化与反序列化并通过网络发送他们，

flink 使用 Type Information 的概念的去处理他们，并为每种类型生成序列化器、解串器和比较器

*flink 还可以通过输入输出自动获取类型信息，进而获取序列化器和反序列化器，但在 lambda 或者范型时需要明确指定

1 Supported Data Types (支持的数据类型)

*应避免使用 kryo 序列化类型，因为它是通用的 序列化器，所以效率并不是很高，flink 提供了向 kryo 预注册的方式来提高效率

(1) Primitives : Java、Scala 的原生的类型如 Int String 等

java scala的原生类型都支持,示例为Long类型的stream

```
val numbers: DataStream[Long] =  
env.fromElements(1L, 2L, 3L, 4L)  
numbers.map( n => n + 1)
```

(2) Java and Scala tuples

```

// scala tuple类型的stream
// DataStream of Tuple2[String, Integer] for Person(name, age)
val persons: DataStream[(String, Integer)] = env.fromElements(
  ("Adam", 17),
  ("Sarah", 23))

// filter for persons of age > 18
persons.filter(p => p._2 > 18)

// java tuple类型的stream
// DataStream of Tuple2<String, Integer> for Person(name, age)
DataStream<Tuple2<String, Integer>> persons = env.fromElements(
  Tuple2.of("Adam", 17),
  Tuple2.of("Sarah", 23));

// filter for persons of age > 18
persons.filter(p -> p.f1 > 18);

tuple的元素的访问
Tuple2<String, Integer> personTuple =
Tuple2.of("Alex", "42");
Integer age = personTuple.getField(1); // age = 42

java 的tuple可以更新字段的value
personTuple.f1 = 42;           // set the 2nd field to 42
personTuple.setField(43, 1); // set the 2nd field to 43

```

(3) Scala case classes

```

scala的case class类型的stream也是支持的
case class Person(name: String, age: Int)

val persons: DataStream[Person] =
env.fromElements(
  Person("Adam", 17),
  Person("Sarah", 23))

// filter for persons with age > 18
persons.filter(p => p.age > 18)

```

(4) POJOs, including classes generated by Apache Avro

POJO 类（我理解就是 java 版本的 case class）的条件

- *必须是 public class。
- *它具有不带任何参数的公共构造函数—默认构造函数。
- *所有字段都是公共的或者可以通过 getter 和 setter 访问， getter 和 setter 函数必须遵循默认的命名结构，例如对于 class Y 的 field x，命名结构为 Y getX () 和 setX (Y x)。
- *所有字段的类型均受 Flink 支持

```
public class Person {  
    // both fields are public  
    public String name;  
    public int age;      java pojo类  
  
    // default constructor is present  
    public Person() {}  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}  
  
DataStream<Person> persons = env.fromElements(  
    new Person("Alex", 42),  
    new Person("Wendy", 23));
```

(5) Some special types

支持数组，列表，映射，枚举

Flink 支持几种特殊类型基本类型和对象 Array 类型。 Java 的 ArrayList, HashMap 和 Enum 类型；和 Hadoop 可写类型。此外，它还提供了 Scala 的 Either, Option 和 Try 类型的类型信息，以及 Flink 的 Either 类型的 Java 版本

2 Creating Type Information for Data Types 为数据类型创建类型信息

第一段：

flink 类型系统的核心类是 TypeInformation，它提供了生成序列化器和比较器的必要信息，当基于某些 key 进行 join 和 groupBy 时，TypeInformation 允许 flink 对 key 的字段是否有效做语义检查

第二段：

当程序被提交执行时，Flink 的类型系统会尝试自动为框架处理的每种数据类型派生 TypeInformation，类型提取器分析所有函数的泛型和返回类型，以获得各自的 TypeInformation 对象，如果想定义自己的类型，并告诉 Flink 如何有效地处理它们。在这种情况下，您需要为特定数据类型生成 TypeInformation

第三段：

flink 为 java 和 scala 提供了两个实体类，它们提供静态方法来指定 TypeInformation 信息

java 的实体类：org.apache.flink.api.common.typeinfo.Types 使用方式如图

scala 的实体类：org.apache.flink.api.scala.typeutils.Types 使用方式如图，scala 使用宏函数生成 TypeInformation 对象，所以访问 type 信息请确保导入 import org.apache.flink.streaming.api.scala._

```
java实体类Types获取TypeInformation
// TypeInformation for primitive types
TypeInformation<Integer> intType = Types.INT;

// TypeInformation for Java Tuples
TypeInformation<Tuple2<Long, String>> tupleType
=
    Types.TUPLE(Types.LONG, Types.STRING);

// TypeInformation for POJOs
TypeInformation<Person> personType =
    Types.POJO(Person.class);
```

java 的实体类

```
scala获取TypeInformation
// TypeInformation for primitive types
val stringType: TypeInformation[String] =
    Types.STRING

// TypeInformation for Scala Tuples
val tupleType: TypeInformation[(Int, Long)] =
    Types.TUPLE[Int, Long]

// TypeInformation for case classes
val caseClassType: TypeInformation[Person] =
    Types.CASE_CLASS[Person]
```

scala 的实体类

3 Explicitly Providing Type Information 明确提供类型信息

小节前言：

在大多数情况下，类型提取器利用反射并分析函数签名和子类信息可以自动推断出类型并声称正确的 TypeInformation，一些特殊情况需要指定，例如：java 的范型擦除，有些情况下 flink 可能不会选择 TypeInformation 来生成最高效的序列化器和反序列化器，因此，您可能需要为应用程序中使用的某些数据类型显式提供 TypeInformation 对象给 Flink，有两种方法可以提供 TypeInformation。首先，可以通过实现 ResultTypeQueryable 接口来扩展函数类以显式提供其返回类型的 TypeInformation

(1) 举例为 map 函数提供显示的类型信息需实现 ResultTypeQueryable 函数 代码示例如图

示例展示了在 map 函数中显示的返回类型的类型信息

```
class Tuple2ToPersonMapper extends
MapFunction[(String, Int), Person]
with ResultTypeQueryable[Person] {

    override def map(v: (String, Int)): Person =
Person(v._1, v._2)

    // provide the TypeInformation for the output data
    // type
    override def getProducedType:
TypeInformation[Person] = Types.CASE_CLASS[Person]
}
```

scala 显示指定 TypeInformation

(2) 在 Java DataStream API 中，当定义数据流时，还可以使用 return()方法显式指定运算符的返回类型 代码示例如图

```
java为map的lambda表达式显示指明TypeInformation
DataStream<Tuple2<String, Integer>> tuples = ...
DataStream<Person> persons = tuples
    .map(t -> new Person(t.f0, t.f1))
    // provide TypeInformation for the map lambda
    // function's return type
    .returns(Types.POJO(Person.class));
java 显示指定 TypeInformation
```

五、Defining Keys and Referencing Fields

定义 key 和引用字段

小节前言：

在之前的小节中某些 tf 要求在 input stream type 上提供 key 或 field。在 Flink 中，未在 input type 中预定义 key，例如在使用 key-value 对的系统中。而是将 key 定义为输入数据的函数。因此，不必定义数据类型来保存 key 和 value，这避免了很多样板代码，下面讨论多种不同的方法来引用 field 和在数据类型上定义 key

1.Field Positions

tuple 可以使用字段位置的方式，定义一个或者多个 key

```
val input: DataStream[(Int, String, Long)] = ...
```

//通过字段位置定义一个 key

```
val keyed = input.keyBy(1)
```

//定义多个 key

```
val keyed2 = input.keyBy(1, 2)
```

2.Field Expressions

*可以使用样例类的字段名，tuple 的位置索引

*嵌套类型的 key 引用 如图

3.Key Selectors

//KeySelector 函数接受的输入项可以是随意计算得出而不一定是 input 流的元素

//如下是选择一个最大值作为 key

```
val input : DataStream[(Int, Int)] = ...
```

```
val keyedStream = input.keyBy(value => math.max(value._1, value._2))
```

i. 嵌套类型指定 key

```
case class Address(  
    address: String,  
    zip: String  
    country: String)  
  
case class Person(  
    name: String,  
    birthday: (Int, Int, Int), // year, month, day  
    address: Address)
```

嵌套类型的key指定

If we want to reference a person's ZIP code, we can use a field expression:

```
| val persons: DataStream[Person] = ...  
| persons.keyBy("address.zip") // key by nested POJO  
|   field
```

It is also possible to nest expressions on mixed types.

The following expression accesses the field of a tuple nested in a POJO:

2 也可以将嵌套表达式用在混合类型上
如下将key指定为嵌套tuple里面的字段

```
persons.keyBy("birthday._1") // key by field of  
nested tuple
```

A full data type can be selected using the wildcard field expression "_" (underscore character):

3 也可以使用通配符表达式_选择所有字段

```
persons.keyBy("birthday._") // key by all fields of  
nested tuple
```

六、Implementing Functions

介绍在 DataStream API 中定义和参数化函数的不同方式

1.Function Classes

*前言：flink 暴露了所有的用户自定义函数的接口和抽象类，例如 MapFunction, FilterFunction, and ProcessFunction

*第一种方式：可以实现函数接口或者继承抽象类的方法，然后把这个实现类传入 map 或者 filter 等 tf 来实现，并实现参数化配置 如图 1

将过滤的关键词参数化

```
val tweets: DataStream[String] = ???  
val flinkTweets = tweets.filter(new  
  KeywordFilter("flink"))  
  
class KeywordFilter(keyWord: String) extends  
  FilterFunction[String] {  
  override def filter(value: String): Boolean =  
  {  
    value.contains(keyWord)  
  }  
}
```

*第二种方式：匿名内部类

注意：flink 使用 java serialization 序列化所有的函数对象，序列化之后会分发给 operator 所有的并行 task 实例，用户定义的函数的所有内容都应该可以被序列化，如存在不可序列化的字段，可以使用 richfun 的 open() method 来初始化不可序列化的字段 或者重写 java 的序列化与反序列化方法

2.Lambda Functions

大多数函数都可以使用 lambda 表达式

3.Rich Functions

*前言：rich 函数比其他的 tf 函数功能更多，有两个方法 open()方法是在 tf 调用前被 task 调用一次，close()方法是在 tf 之后被每一个 task 调用一次。详情可以查看右方配图 2，rich 函数主要解决无法序列化的变量，广播参数等

```

rich fun示例
class MyFlatMap extends RichFlatMapFunction[Int,
(Int, Int)] {
    var subTaskIndex = 0
    override def open(configuration: Configuration): Unit = {
        subTaskIndex =
getRuntimeContext.getIndexOfThisSubtask
        // do some initialization
        // e.g., establish a connection to an external system
    }

    override def flatMap(in: Int, out: Collector[(Int,
Int)]): Unit = {
        // subtasks are 0-indexed
        if(in % 2 == subTaskIndex) {
            out.collect((subTaskIndex, in))
        }
        // do some more processing
    }

    override def close(): Unit = {
        // do some cleanup, e.g., close connections to
external systems
    }
}

```

dataset中可以配置

用于检索task信息，例如
函数并行度、subtask
index、执行fun的task
name、访问分区状态

图 2 rich funcation 代码示例

七、Including External and Flink Dependencies

外部及 flink 依赖

前言：应用程序的运行普遍需要第三方依赖或者 flink 内部特殊的依赖例如 cep、table api、sql 等，flink cluster 默认只加载核心 API (dataStream 和 dataSet)，其他的依赖需要明确提供

1. flink 尽量保持低的依赖，因为很多的问题是由于版本冲突造成的，避免版本冲突有两种方式
 - (1)一种是使用 fat jar，将依赖全部包含在 jar 文件中
 - (2)另一种是将依赖添加到 flink 的./lib 文件下，这种方式可以将依赖应用给所有的程序，但也可能会发生干扰，比较好的方式还是使用 maven 打成 fat jar

八、Summary 总结

在本章中，我们介绍了 Flink 的 DataStream API 的基础。 我们查看了 Flink 程序的结构，并了解了如何结合数据和分区 tf 来构建流应用程序。 还研究了受支持的数据类型以及指定 key 和用户自定义函数的不同方法。 如果仔细琢磨例子，则希望接下来有更好的想法，在第 6 章中，事情将变得更加有趣-我们将学习如何使用 window operator 和时间语义来丰富程序