第 3 章 DataStream API

学习目标

- 熟悉 DataStream 程序开发流程，能够说出开发 DataStream 程序的 5 个步骤。
- 了解 DataStream 的数据类型，能够说出 DataStream 程序常用的数据类型。
- 熟悉执行环境，能够在 DataStream 程序中创建和配置执行环境。
- 掌握数据输入，能够灵活运用 DataStream API 提供的数据源算子读取数据。
- 掌握数据转换，能够灵活运用 DataStream API 提供的转换算子对 DataStream 对象进行转换。
- 掌握数据输出，能够灵活运用 DataStream API 提供的接收器算子输出 DataStream 对象的数据。
- 掌握应用案例——词频统计，能够独立实现词频统计的 DataStream 程序。

Flink 作为一种实时计算的流处理框架，它提供了 DataStream API 支持原生流处理，通过 DataStream API 可以让开发者灵活且高效地编写 Flink 程序。不过 DataStream API 在设计之初，仅用于实现处理无界流的 Flink 程序，即流处理程序。随着 Flink 逐渐向批流一体靠拢，DataStream API 同时兼顾了实现处理有界流的 Flink 程序，即批处理程序。本章针对 DataStream API 实现 Flink 程序的相关知识进行讲解。

3.1 DataStream 程序的开发流程

DataStream API 允许在 Flink 中实现流处理程序，这些程序称为 DataStream 程序。DataStream 程序遵循 Flink 程序结构，包括 Source、Transformation 和 Sink 三部分，这三部分共同构成了程序的执行逻辑。然而，仅凭执行逻辑还不足以让 DataStream 程序正常运行。需要在程序中创建执行环境（Execution Environment）并添加执行器（Execute）来触发程序执行。

DataStream 程序开发流程的详细介绍如下。

1. 创建执行环境

实现 DataStream 程序的第一步是创建执行环境。执行环境类似于程序的说明书，它告知 Flink 如何解析和执行 DataStream 程序。

2. 读取数据源

在创建执行环境之后，需要通过读取数据源获取数据。数据输入来源通常称为数据源。

3. 定义数据转换

在从数据源获取数据之后，根据实际业务逻辑对读取的数据定义各种转换操作。数据转

换是流处理过程中的重要环节，通过对读取的数据进行转换操作，可以对数据进行清洗、加工、重组等处理，以适应实际的业务需求。

4. 输出计算结果

数据经过转换后的最终结果将被写入外部存储，以便为外部应用提供支持。

5. 添加执行器

DataStream 程序开发的最后一步是添加执行器，执行器负责实际触发读取数据源、数据转换和输出计算结果的操作。默认情况下，这些操作仅在作业中定义并添加到数据流图中，并不会实际执行。

在 DataStream 程序中添加执行器的过程相对简单，下面通过示例进行简要说明。在 DataStream 程序中，执行器的添加通过调用执行环境的 execute()方法来实现，同时可以向这个方法传递一个参数以指定作业的名称，其示例代码如下。

```
executionEnvironment.execute("itcast");
```

上述示例代码中，executionEnvironment 为 DataStream 程序的执行环境，itcast 为指定的作业名称。需要注意的是，尽管在大多数情况下，DataStream 程序按照上述 5 个步骤进行开发，但在某些特殊场景中，可以跳过定义数据转换这一步。例如，在调试 DataStream 程序中读取数据源的操作时，可以不定义数据转换，而是直接输出读取到的数据。

3.2 DataStream 的数据类型

DataStream 是 Flink 中用于表示数据集合的类，它是一种抽象的数据结构，实现的 DataStream 程序其实就是基于这个类的处理，通过在类中使用泛型来描述数据集合中每个元素的数据类型。

DataStream 支持多种数据类型，可以方便地处理不同结构的数据，其中常用的数据类型包括基本数据类型、元组(Tuple)类型和 POJO 类型，具体介绍如下。

1. 基本数据类型

DataStream 支持 Java 和 Scala 的基本数据类型，如整数、浮点数、字符串等。例如，定义 DataStream 的数据类型为字符串的示例代码如下。

```
DataStream<String> stringStream = env.fromElements("A", "B", "C", "D");
```

上述示例代码中，env 为 DataStream 程序的执行环境，fromElements()方法为 DataStream API 提供的预定义数据源算子。

2. 元组类型

Flink 中的元组(Tuple)是一种特殊的数据类型，用于封装不同类型的元素。Flink 支持的元组类型有 Tuple1 到 Tuple25，分别表示包含 1～25 个元素的元组。例如，定义 DataStream 的数据类型为元组的示例代码如下。

```
DataStream<Tuple2<String, Integer>> tupleStream =
env.fromElements(Tuple2.of("A", 1), Tuple2.of("B", 2));
```

上述示例代码中，元组的类型为 Tuple2，表示元组具有两个元素，其中第一个元素的数据类型为 String(字符串)，第二个元素的数据类型为 Integer(整数)。

3. POJO 类型

POJO(Plain Old Java Object)是一个符合特定条件的 Java 类,它用于表示具有多个属性的数据结构。在定义数据类型为 POJO 的 DataStream 时,需要注意以下几点。

(1) POJO 必须是公共的。

(2) POJO 具有公共的无参构造方法。

(3) POJO 的属性可以是私有的,也可以是公共的。如果属性是私有的,那么必须具有 getter()和 setter()方法。

例如,定义 DataStream 的数据类型为 POJO 的示例代码如下。

```
public class Person {
    private String name;
    private int age;
    public Person() {
    }
    public Person(String name,int age) {
        this.name = name;
        this.age = age;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
DataStream<Person> personStream =
env.fromElements(new Person("Alice", 30), new Person("Bob", 25));
```

上述示例代码中,Person 为定义的 POJO,它具有两个属性 name 和 age,其中属性 name 的数据类型为 String,属性 age 的数据类型为 int。

3.3　执行环境

执行环境在 DataStream 程序中扮演着至关重要的角色,它负责任务调度、资源分配以及程序的执行。因此,在实现 DataStream 程序时,首先需要创建一个合适的执行环境,并且基于执行环境配置 DataStream 程序。同样,在工作和学习中,我们都应当考虑到个体的实际情况,寻找最适合个体的方法和策略,而非盲目跟随或者一刀切。

DataStream API 提供了一个 StreamExecutionEnvironment 类用于创建和配置执行环境,具体内容如下。

1. 创建执行环境

StreamExecutionEnvironment 类提供了 3 种方法用于创建执行环境,它们分别是 createLocalEnvironment()、createRemoteEnvironment()和 getExecutionEnvironment(),具

体介绍如下。

1）createLocalEnvironment()

该方法创建的执行环境为本地执行环境，它会在本地计算机创建一个本地环境来执行 DataStream 程序，通常用于在 IDE（集成开发环境）内部执行 DataStream 程序。

使用 createLocalEnvironment()方法创建执行环境的示例代码具体如下。

```
StreamExecutionEnvironment localEnvironment =
        StreamExecutionEnvironment.createLocalEnvironment();
```

2）createRemoteEnvironment()

该方法创建的执行环境为远程执行环境，它会将 DataStream 程序提交到指定的 Flink 执行，使用该方法时需要依次传入参数 host 和 port，它们分别用于指定 Flink Web UI 的 IP 和端口号。

使用 createRemoteEnvironment()方法创建执行环境的示例代码具体如下。

```
StreamExecutionEnvironment remoteEnvironment =
        StreamExecutionEnvironment.createRemoteEnvironment(
                "192.168.121.144",
                8081
            );
```

上述代码中，192.168.121.144 和 8081 分别表示 Flink Web UI 的 IP 和端口号。

3）getExecutionEnvironment()

该方法创建的执行环境基于运行 DataStream 程序的环境，如果 DataStream 程序在 IDE 运行，那么创建的执行环境为本地执行环境。如果 DataStream 程序被提交到 Flink 运行，那么创建的执行环境为远程执行环境。

使用 getExecutionEnvironment()方法创建执行环境的示例代码具体如下。

```
StreamExecutionEnvironment executionEnvironment =
          StreamExecutionEnvironment.getExecutionEnvironment();
```

需要说明的是，使用 getExecutionEnvironment()方法创建执行环境的方式较为常用，因为它能够根据 DataStream 程序的运行环境自动创建本地执行环境或远程执行环境，使用起来较为便捷，后续实现的 DataStream 程序主要使用 getExecutionEnvironment()方法创建执行环境。

2. 配置执行环境

StreamExecutionEnvironment 类提供了多个方法用于配置 DataStream 程序，这里介绍基础且常用的两个方法 setRuntimeMode()和 setParallelism()，具体内容如下。

1）setRuntimeMode()

该方法用于配置 DataStream 程序的执行模式，DataStream 程序支持两种执行模式，分别是流处理（STREAMING）和批处理（BATCH），其中流处理是 DataStream 程序默认使用的执行模式，用于实时处理无界流；批处理是专门用于处理有界流的执行模式。除此之外，DataStream 程序还支持通过自行判断选择使用的执行模式，称为自动模式，自动模式根据读取数据源的数据是否有界，自行选择 DataStream 程序使用的执行模式为流处理还是批处理。

由于 DataStream 程序默认使用的执行模式为流处理，所以这里重点介绍指定批处理或自

动模式的执行模式，在 DataStream 程序中，可以调用执行环境的 setRuntimeMode()方法显式指定执行模式为批处理或自动模式，具体示例代码如下。

```
//指定执行模式为批处理
executionEnvironment.setRuntimeMode(RuntimeExecutionMode.BATCH);
//指定执行模式为自动模式
executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
```

2）setParallelism()

该方法用于配置 DataStream 程序的并行度，如指定 DataStream 程序的并行度为 3 的示例代码如下。

```
executionEnvironment.setParallelism(3);
```

需要说明的是，基于执行环境配置的并行度，其优先级要高于配置文件 flink-conf.yaml 和 flink 命令指定的并行度。除此之外，setParallelism()方法还可以为 DataStream 程序中的不同算子单独配置并行度，其优先级要高于执行环境配置的并行度。

综上所述，不同方式配置并行度的优先级从高到低依次为算子配置的并行度、执行环境配置的并行度、flink 命令配置的并行度，配置文件配置的并行度。

【注意】 算子配置的并行度会受到自身具体实现的约束，如 socketTextStream 是一个非并行的数据源算子，此时配置的并行度对该算子是无效的。

3.4　数据输入

要处理数据，就必须先获取数据。因此，在创建执行环境后，首要任务就是将数据从数据源读取到 DataStream 程序中。DataStream 程序可以从各种数据源中读取数据，并通过 DataStream API 提供的算子将其转换为 DataStream 对象，这些算子称为数据源算子（Source Operator），可以看作 StreamExecutionEnvironment 类提供的一类方法。

如果想要从文件、Socket 或集合读取数据，那么可以直接使用 DataStream API 提供的预定义数据源算子即可。如果想要从外部系统读取数据，如 Kafka、RabbitMQ 等，或者读取自定义 Source 的数据，那么可以使用 DataStream API 提供的数据源算子 addSource 来实现。本节针对 DataStream 程序的数据输入进行详细讲解。

3.4.1　从集合读取数据

DataStream API 提供了预定义数据源算子 fromCollection 和 fromElements，用于从集合读取数据，具体介绍如下。

1. fromCollection

使用预定义数据源算子 fromCollection 从集合读取数据时，需要传递一个集合类型的参数，其语法格式如下。

```
fromCollection(collection)
```

上述语法格式中，collection 用于指定集合。

2. fromElements

使用预定义数据源算子 fromElements 从集合读取数据时，可以传递任意数量的 Java 对

象作为参数，Java 对象可以是集合类型、数组类型、字符串类型等，其语法格式如下。

```
fromElements(T,T,T,...)
```

上述语法格式中，T,T,T,...表示给定的对象序列，对象序列中每个 T 表示一个 Java 对象。需要注意的是，对象序列中所有 Java 对象的类型必须相同。

接下来通过一个案例演示如何从集合读取数据。在实现本案例之前先说明相关的环境要求，本书使用 IntelliJ IDEA 作为集成开发环境，并且使用 JDK 8 构建 Java 运行环境，因此希望读者在实现 DataStream 程序之前，确保计算机中安装了 IntelliJ IDEA 和 JDK 8。为了后续知识讲解便利，这里将分步骤讲解本案例的实现过程，具体步骤如下。

1）创建 Java 项目

在 IntelliJ IDEA 中基于 Maven 创建 Java 项目，指定项目使用的 JDK 为本地安装的 JDK 8，以及指定项目名称为 Flink_Chapter03。Flink_Chapter03 项目创建完成后的效果如图 3-1 所示。

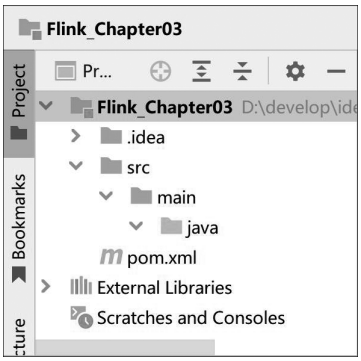

图 3-1　Flink_Chapter03 项目创建完成后的效果

2）构建项目目录结构

在 Java 项目的 java 目录中创建包 cn.datastream.demo 用于存放实现 DataStream 程序的类。

3）添加依赖

在 Java 项目的 pom.xml 文件中添加依赖，依赖添加完成的效果如文件 3-1 所示。

文件 3-1　pom.xml

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2  <project xmlns = "http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
5  http://maven.apache.org/xsd/maven-4.0.0.xsd">
6       <modelVersion>4.0.0</modelVersion>
7       <groupId>cn.itcast</groupId>
8       <artifactId>Flink_Chapter03</artifactId>
9       <version>1.0-SNAPSHOT</version>
10      <properties>
11          <maven.compiler.source>8</maven.compiler.source>
12          <maven.compiler.target>8</maven.compiler.target>
13          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14  </properties>
```

```
15 <dependencies>
16     <dependency>
17         <groupId>org.apache.flink</groupId>
18         <artifactId>flink-streaming-java</artifactId>
19         <version>1.16.0</version>
20     </dependency>
21     <dependency>
22         <groupId>org.apache.flink</groupId>
23         <artifactId>flink-clients</artifactId>
24         <version>1.16.0</version>
25     </dependency>
26 </dependencies>
27 </project>
```

在文件 3-1 中，第 15～26 行代码为添加的内容，其中第 16～20 行代码表示 DataStream API 的核心依赖；第 21～25 行代码表示 Flink 客户端依赖。

依赖添加完成后，确认添加的依赖是否存在于 Java 项目中，在 IntelliJ IDEA 主界面的右侧单击 Maven 选项卡展开 Maven 窗口，在 Maven 窗口单击 Dependencies 折叠框，如图 3-2 所示。

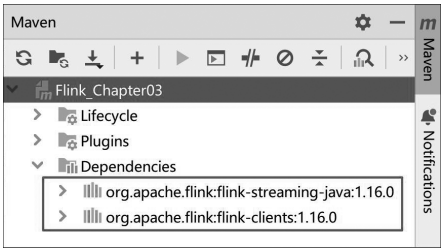

图 3-2　查看添加的依赖

从图 3-2 可以看出，依赖已经成功添加到 Java 项目中，如果这里未显示添加的依赖，则可以单击 按钮重新加载 pom.xml 文件。

4）实现 DataStream 程序

创建一个名为 ReadCollectionDemo 的 DataStream 程序，该程序能够从集合中读取数据并将其输出到控制台，具体代码如文件 3-2 所示。

文件 3-2　ReadCollectionDemo.java

```
1  public class ReadCollectionDemo {
2      public static void main(String[] args) throws Exception {
3          //创建执行环境
4          StreamExecutionEnvironment executionEnvironment =
5                  StreamExecutionEnvironment.getExecutionEnvironment();
6          //指定执行模式为自动模式
7          executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
8          List<Tuple2<String,Integer>> list = new ArrayList();
9          list.add(new Tuple2<String, Integer>("user01", 23));
10         list.add(new Tuple2<String, Integer>("user02", 25));
11         list.add(new Tuple2<String, Integer>("user03", 26));
```

```
        DataStream<Tuple2<String,Integer>> fromCollectionDataStream =
                executionEnvironment.fromCollection(list);
        DataStream<Tuple2<String, Integer>> fromElementsDataStream =
                executionEnvironment.fromElements(
                new Tuple2<String, Integer>("user04", 22),
                new Tuple2<String, Integer>("user05", 23),
                new Tuple2<String, Integer>("user06", 27));
        fromCollectionDataStream.print("fromCollectionDataStream 的数据");
        fromElementsDataStream.print("fromElementsDataStream 的数据");
        executionEnvironment.execute();
    }
}
```

上述代码中，第 8～11 行代码创建集合 list，并且向集合中插入 3 个元素。第 12、13 行代码使用预定义数据源算子 fromCollection 从集合 list 读取数据，并将其转换为 DataStream 对象 fromCollectionDataStream，该对象的数据类型与集合 list 中元素的数据类型一致。

第 14～18 行代码使用预定义数据源算子 fromElements 从类型为 Tuple2 的对象序列读取数据，并将其转换为 DataStream 对象 fromElementsDataStream，该对象的数据类型与对象序列中每个对象的类型一致。

第 19、20 行代码使用 DataStream API 提供的预定义输出算子 print，分别将 DataStream 对象 fromCollectionDataStream 和 fromElementsDataStream 的数据输出到控制台。

文件 3-2 的运行结果如图 3-3 所示。

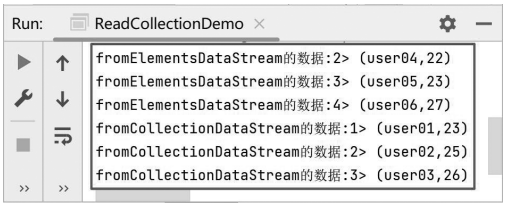

图 3-3　文件 3-2 的运行结果

从图 3-3 可以看出，fromCollectionDataStream 的数据与集合 list 的数据相同，fromElementsDataStream 的数据与对象序列的数据相同。因此说明，成功使用了预定义数据源算子 fromCollection 和 fromElements 从集合读取数据。

3.4.2　从文件读取数据

DataStream API 提供了预定义数据源算子 readTextFile，用于从指定文件系统的文件读取数据，如本地文件系统、HDFS 等，其语法格式如下。

```
readTextFile(path)
```

上述语法中，参数 path 用于指定文件的目录。

接下来通过一个案例来演示如何从文件读取数据。本案例将分别从本地文件系统的 D:\FlinkData\Flink_Chapter03 目录和 HDFS 的/FlinkData/Flink_Chapter03 目录读取文件 Person.csv 和 Person。关于这两个文件的内容如图 3-4 所示。

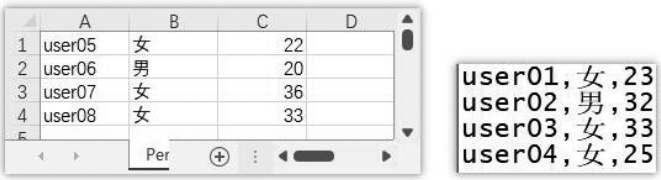

图 3-4　文件 Person.csv 和 Person 的内容

创建一个名为 ReadFileDemo 的 DataStream 程序，该程序能够从文件 Person.csv 和 Person 中读取数据并将其输出到控制台，具体代码如文件 3-3 所示。

文件 3-3　ReadFileDemo.java

```
1  public class ReadFileDemo {
2      public static void main(String[] args) throws Exception {
3          StreamExecutionEnvironment executionEnvironment =
4                  StreamExecutionEnvironment.getExecutionEnvironment();
5        executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
6          DataStream<String> HDFSFileDataStream =
7                  executionEnvironment.
8                          readTextFile("hdfs://192.168.121.144:9820/" +
9                                  "FlinkData/Flink_Chapter03/Person");
10         DataStream<String> LocalFileDataStream =
11                 executionEnvironment
12                 .readTextFile("D:\\FlinkData\\Flink_Chapter03\\Person.csv");
13         HDFSFileDataStream.print("HDFSFileDataStream 的数据");
14         LocalFileDataStream.print("LocalFileDataStream 的数据");
15         executionEnvironment.execute();
16     }
17 }
```

上述代码中，第 6～9 行代码使用预定义数据源算子 readTextFile 从文件 Person 读取数据，并将其转换为 DataStream 对象 HDFSFileDataStream。第 10～12 行代码使用预定义数据源算子 readTextFile 从文件 Person.csv 读取数据，并将其转换为 DataStream 对象 LocalFileDataStream。

由于文件 3-3 实现的 DataStream 程序从 HDFS 的文件读取数据，所以在运行文件 3-3 之前，需要在 Java 项目的依赖管理文件 pom.xml 的＜dependencies＞标签中添加 Hadoop 客户端依赖，具体内容如下。

```
1  <dependency>
2      <groupId>org.apache.hadoop</groupId>
3      <artifactId>hadoop-client</artifactId>
4      <version>3.2.2</version>
5  </dependency>
```

确保 Hadoop 处于启动状态，以及上述依赖成功添加到 Java 项目之后，文件 3-3 的运行结果如图 3-5 所示。

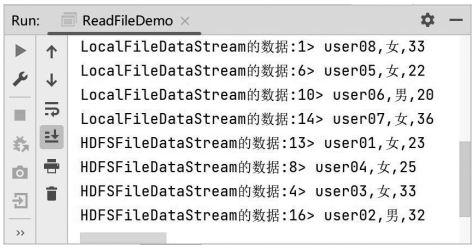

图 3-5 文件 3-3 的运行结果

从图 3-5 可以看出，HDFSFileDataStream 的数据与文件 Person 的数据相同，LocalFileDataStream 的数据与文件 Person.csv 的数据相同。因此说明，成功使用了预定义数据源算子 readTextFile 从文件读取数据。

3.4.3 从 Socket 读取数据

DataStream API 提供了预定义数据源算子 socketTextStream，用于从 Socket 读取数据，其语法格式如下。

```
socketTextStream(hostname,port)
```

上述语法格式中，参数 hostname 用于指定 Socket 的主机名或 IP 地址；port 用于指定 Socket 的端口号。

接下来通过一个案例来演示如何从 Socket 读取数据，具体操作步骤如下。

（1）本案例通过网络工具 Ncat 建立 TCP 连接，以此来演示如何从 Socket 获取数据。在虚拟机 Flink01 执行如下命令在线安装 Ncat。

```
$ yum install -y nc
```

（2）在虚拟机 Flink01 中通过 Ncat 工具建立 TCP 连接，并指定端口号为 9999，具体命令如下。

```
$ nc -lk 9999
```

上述命令执行完成后的效果如图 3-6 所示。

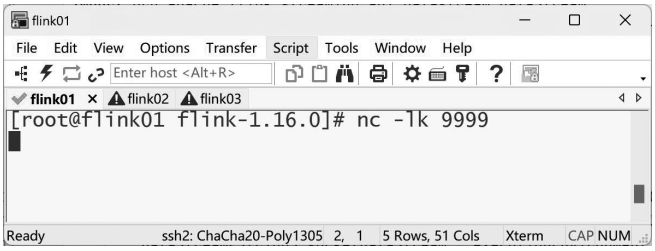

图 3-6 建立 TCP 连接

在图 3-6 中成功建立了 TCP 连接，并等待发送数据。

（3）创建一个名为 ReadSocketDemo 的 DataStream 程序，该程序能够读取 TCP 连接发送的数据并将其输出到控制台，具体代码如文件 3-4 所示。

文件 3-4　ReadSocketDemo.java

```
1  public class ReadSocketDemo {
2    public static void main(String[] args) throws Exception {
3      StreamExecutionEnvironment executionEnvironment =
4              StreamExecutionEnvironment.getExecutionEnvironment();
5      executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
6      DataStream<String> socketDataStream = executionEnvironment
7              .socketTextStream("192.168.121.144", 9999);
8      socketDataStream.print("socketDataStream 的数据");
9      executionEnvironment.execute();
10    }
11 }
```

第 6、7 行代码使用预定义数据源算子 socketTextStream 从虚拟机 Flink01 的 TCP 连接读取数据，并将其转换为 DataStream 对象 socketDataStream。

文件 3-4 的运行效果(1)如图 3-7 所示。

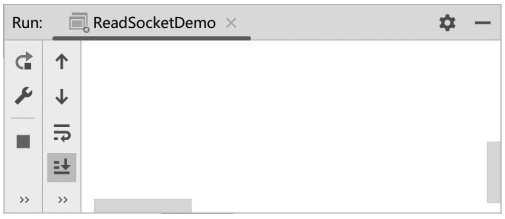

图 3-7　文件 3-4 的运行效果(1)

在图 3-7 中，当文件 3-4 运行完成后，会等待虚拟机 Flink01 的 TCP 连接发送数据。

在图 3-6 所示的界面输入“apple,banana,pear”之后按 Enter 键发送数据，此时查看图 3-7 所示界面，文件 3-4 的运行效果(2)如图 3-8 所示。

图 3-8　文件 3-4 的运行效果(2)

从图 3-8 可以看出，socketTextStream 的数据与 TCP 连接发送的数据相同。因此说明，成功使用了预定义数据源算子 socketTextStream 从 Socket 读取数据。

3.4.4　从 Kafka 读取数据

DataStream API 提供了数据源算子 addSource 用于从外部系统读取数据，其语法格式如下。

```
addSource(sourcefunction)
```

上述语法格式中，sourcefunction 用于指定实现连接外部系统的方式，不同外部系统的实现方式有所不同。这里以常用的外部系统 Kafka 为例进行讲解。

Kafka 是由 Apache 软件基金会开发的一个开源流处理平台，它是一种高吞吐量的分布式发布订阅消息系统。在实时流处理的应用场景中，通常将 Kafka 和 Flink 进行结合使用，其中

Kafka 负责数据的收集和传输，Flink 负责从 Kafka 读取数据进行计算。这种结合使用不同技术的思维方式提醒人们在面对复杂的问题和挑战时，需要将来自不同领域的知识、技能和资源进行有效整合，通过跨界合作和交叉思维来发掘更多可能的解决方案。这种思维方式有助于我们打破独立思考的限制，拓宽视野，发现更多创新的思路和方法。

接下来通过一个案例来演示如何从 Kafka 读取数据，具体操作步骤如下。

1. 安装 Kafka

实现本案例的首要任务便是安装 Kafka，本书使用 Kafka 的版本为 3.3.0，这里以虚拟机 Flink01 为例，演示如何安装 Kafka，具体操作步骤如下。

1）上传 Kafka 安装包

使用 rz 命令将本地计算机中准备好的 Kafka 安装包 kafka_2.12-3.3.0.tgz 上传到虚拟机的/export/software/目录。

2）安装 Kafka

使用 tar 命令将 Kafka 安装到虚拟机的/export/servers/目录，具体命令如下。

```
$ tar -zxvf /export/software/kafka_2.12-3.3.0.tgz -C /export/servers/
```

3）启动 Kafka 内置的 ZooKeeper

Kafka 的运行与 ZooKeeper 有着密不可分的关系，如 Kafka 通过 ZooKeeper 来存储元数据。在实际应用场景中，Kafka 通常基于独立部署的 ZooKeeper 来运行，不过出于便捷性考虑，本案例使用的 Kafka 是基于其内置的 ZooKeeper 来运行的。

为了避免端口号冲突，需要提前关闭第 2 章在虚拟机 Flink01 中启动的 Zookeeper 服务后，在虚拟机的/export/servers/kafka_2.12-3.3.0 目录执行如下命令启动 Kafka 内置 ZooKeeper。

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

上述命令通过 Kafka 提供的脚本文件 kafka-server-start.sh 启动 Kafka 内置 ZooKeeper。Kafka 内置 ZooKeeper 启动完成的效果如图 3-9 所示。

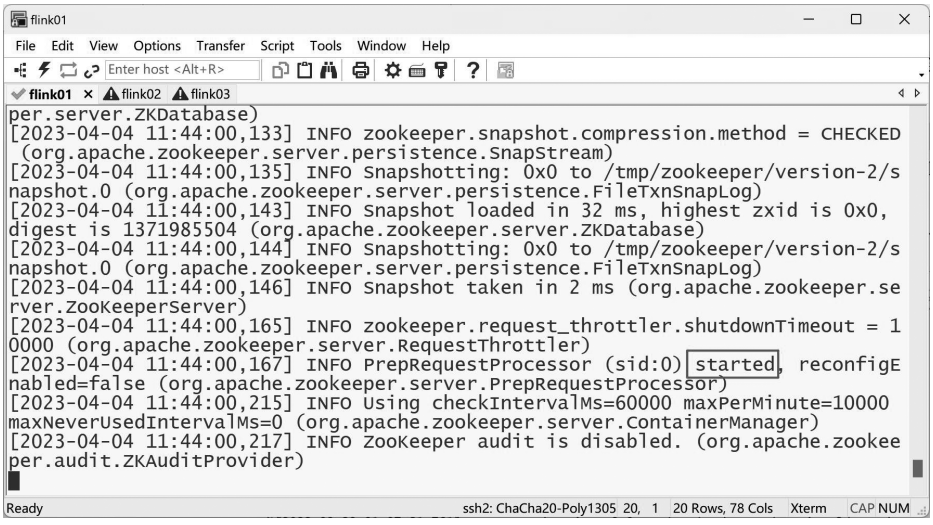

图 3-9 Kafka 内置 ZooKeeper 启动完成的效果

在图 3-9 中，若启动信息中出现 started，则证明 Kafka 内置 ZooKeeper 启动成功。需要注意的是，Kafka 内置 ZooKeeper 启动完成后，会占用当前操作窗口，用户无法在当前窗口进行其他操作。如果关闭该窗口，Kafka 内置 ZooKeeper 会停止运行。

4）启动 Kafka

启动 Kafka 的本质是在虚拟机中启动一个消息代理服务（Broker Service），消息代理服务是 Kafka 运行的依据。在 SecureCRT 中，为虚拟机 Flink01 克隆一个新的操作窗口用于启动 Kafka。

在虚拟机的/export/servers/kafka_2.12-3.3.0 目录执行如下命令启动 Kafka。

```
$ bin/kafka-server-start.sh config/server.properties
```

上述命令通过 Kafka 提供的脚本文件 kafka-server-start.sh 启动 Kafka。Kafka 启动完成的效果如图 3-10 所示。

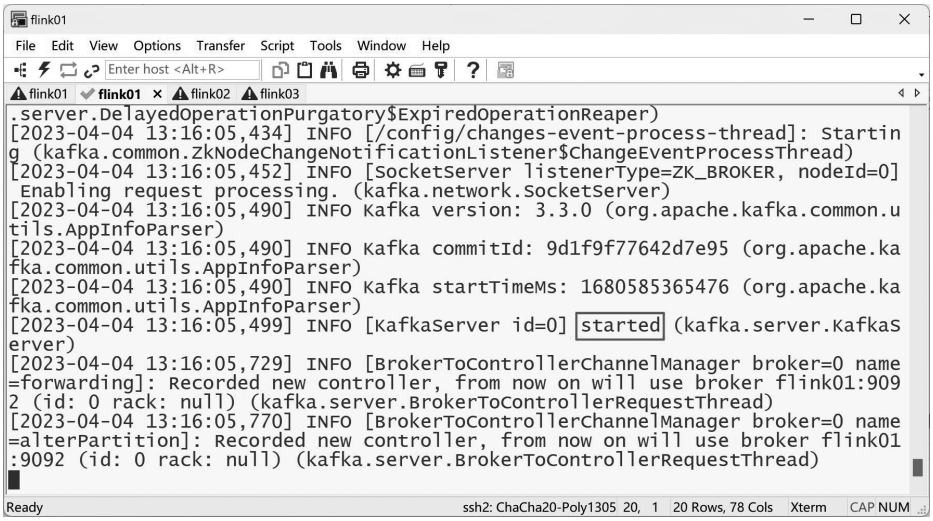

图 3-10　Kafka 启动完成的效果

在图 3-10 中，若启动信息中出现 started，则证明 Kafka 启动成功。需要注意的是，Kafka 启动完成后，会占用当前操作窗口，用户无法在当前窗口进行其他操作。如果关闭该窗口，Kafka 会停止运行。

5）测试 Kafka

测试 Kafka 的主要目的是验证 Kafka 生产者（producer）向指定主题（topic）发布的消息，是否可以被订阅该主题的 Kafka 消费者（consumer）所接收，具体实现过程如下。

（1）在 SecureCRT 中，为虚拟机 Flink01 克隆一个新的操作窗口用于创建主题。在虚拟机的/export/servers/kafka_2.12-3.3.0 目录执行如下命令创建主题。

```
$ bin/kafka-topics.sh --create --topic kafka-source-topic \
--bootstrap-server 192.168.121.144:9092
```

上述命令通过 Kafka 提供的脚本文件 kafka-topics.sh 操作主题，其中参数--create 用于创建主题；参数--topic 用于指定主题名称，这里指定的主题名称为 kafka-source-topic；参数--bootstrap-server 用于指定 Kafka 的 IP 地址和端口号，Kafka 默认的端口号是 9092。主题创

建完成的效果如图 3-11 所示。

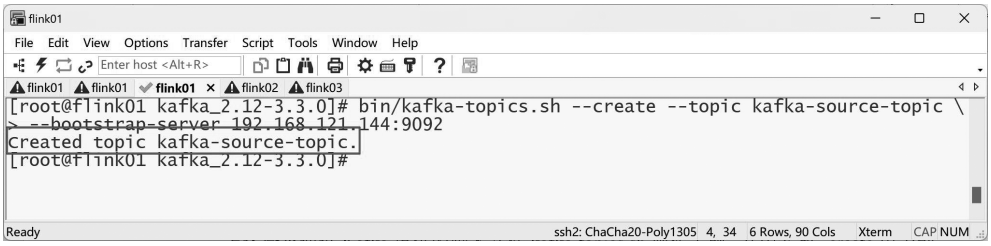

图 3-11 主题创建完成的效果

在图 3-11 中，若主题创建完成后出现“Created topic kafka-source-topic.”，则证明成功在 Kafka 创建名称为 kafka-source-topic 的主题。

（2）启动 Kafka 生产者，该生产者向主题 kafka-source-topic 发布消息，在/export/servers/kafka_2.12-3.3.0 目录执行如下命令。

```
$ bin/kafka-console-producer.sh --topic kafka-source-topic \
--bootstrap-server 192.168.121.144:9092
```

上述命令通过 Kafka 提供的脚本文件 kafka-console-producer.sh 启动 Kafka 生产者，其中参数--topic 用于指定主题名称；参数--bootstrap-server 用于指定 Kafka 的 IP 地址和端口号。Kafka 生产者启动完成的效果如图 3-12 所示。

图 3-12 Kafka 生产者启动完成的效果

在图 3-12 中，当 Kafka 生产者启动完成后，便可以在“>”位置输入要发布到主题 kafka-source-topic 的消息。

（3）在 SecureCRT 中，为虚拟机 Flink01 克隆一个新的操作窗口来启动 Kafka 消费者，该消费者通过订阅主题 kafka-source-topic 接收 Kafka 生产者发布的消息。

在虚拟机的/export/servers/kafka_2.12-3.3.0 目录执行如下命令启动 Kafka 消费者。

```
$ bin/kafka-console-consumer.sh --topic kafka-source-topic \
--from-beginning --bootstrap-server 192.168.121.144:9092
```

上述命令通过 Kafka 提供的脚本文件 kafka-console-consumer.sh 启动 Kafka 消费者，其中参数--topic 用于指定主题名称；参数--bootstrap-server 用于指定 Kafka 的 IP 地址和端口号。参数--from-beginning 表示 Kafka 消费者从主题的第一条消息开始消费。Kafka 消费者启动完成的效果如图 3-13 所示。

在图 3-13 中，当 Kafka 消费者启动完成后，会等待 Kafka 生产者向主题 kafka-source-topic 发布消息。

（4）在 Kafka 生产者输入“This is my first event”之后按 Enter 键发布消息，此时，查看

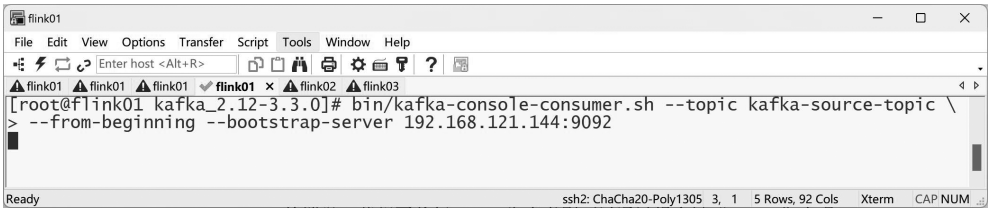

图 3-13　Kafka 消费者启动完成的效果

Kafka 消费者是否可以接收消息，如图 3-14 所示。

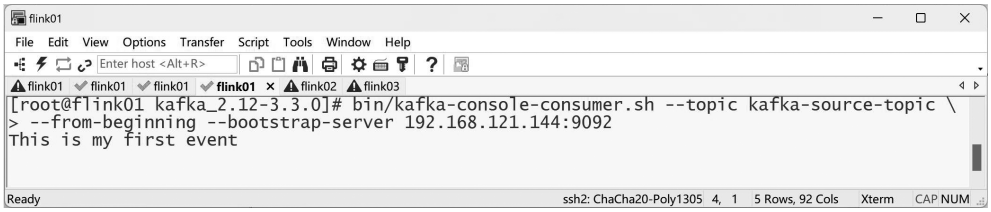

图 3-14　发布消息

从图 3-14 可以看出，Kafka 消费者可以成功接收到 Kafka 生产者向主题 kafka-source-topic 发布的消息 This is my first event，因此说明成功安装 Kafka。

需要说明的是，若想要关闭 Kafka 生产者或消费者，则可以通过组合键 Ctrl+C 实现。

2. 添加依赖

Flink 提供了与 Kafka 建立连接的连接器，但是需要通过添加依赖才能使用。在 Java 项目的依赖管理文件 pom.xml 的<dependencies>标签中添加以下内容，即添加与 Kafka 建立连接的连接器依赖。

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka</artifactId>
    <version>1.16.0</version>
</dependency>
```

3. 实现 DataStream 程序

创建一个名为 ReadKafkaDemo 的 DataStream 程序，该程序能够从 Kafka 读取数据并将其输出到控制台，具体代码如文件 3-5 所示。

文件 3-5　ReadKafkaDemo.java

```
public class ReadKafkaDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment executionEnvironment =
                StreamExecutionEnvironment.getExecutionEnvironment();
        executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
        Properties properties = new Properties();
        properties.setProperty("bootstrap.servers","192.168.121.144:9092");
        properties.setProperty("group.id","kafkasource");
        DataStream<String> kafkaDataStream =
                executionEnvironment.addSource(
                        new FlinkKafkaConsumer<>("kafka-source-topic",
```

```
12                                  new SimpleStringSchema(),
13                                  properties));
14          kafkaDataStream.print("kafkaDataStream 的数据");
15          executionEnvironment.execute();
16      }
17  }
```

上述代码中，第 6～8 行代码用于指定 Kafka 的配置信息，其中第 7 行代码用于指定 Kafka 的 IP 地址和端口号；第 8 行代码用于指定 Kafka 消费者的 GroupId。

第 9～13 行代码使用数据源算子 addSource 从 Kafka 读取数据，并将其转换为 DataStream 对象 kafkaDataStream。使用数据源算子 addSource 从 Kafka 读取数据时，需要在数据源算子 addSource 中实例化 FlinkKafkaConsumer 类的同时传递三个参数，其中第一个参数用于指定 Kafka 主题名称；第二个参数负责将接收的消息反序列化为字符串；第三个参数用于指定 Kafka 的配置信息。

4. 测试 DataStream 程序

为了避免在运行文件 3-5 时，DataStream 程序将 IP 地址重定向至主机名，导致无法连接到 Kafka，可以在本地计算机的 hosts 文件中添加虚拟机 Flink01 的主机名和 IP 地址的映射。

在虚拟机 Flink01 分别启动 Kafka 内置 ZooKeeper、Kafka 和 Kafka 生产者，其中 Kafka 生产者指定的主题名称需要与文件 3-5 中指定的主题名称一致。

图 3-15　此时查看文件 3-5 的运行结果

运行文件 3-5，在 Kafka 生产者发布一条消息 itcast，此时查看文件 3-5 的运行结果，如图 3-15 所示。

从图 3-15 可以看出，kafkaDataStream 的数据与 Kafka 生产者发布的消息一致。因此说明，成功使用了数据源算子 addSource 从 Kafka 读取数据。

3.4.5　自定义 Source

如果在实际使用过程中遇到特殊需求，如需要从外部系统读取数据源，但是 Flink 没有提供相应的连接器，或者预定义的数据源算子无法满足实际需求，那么可以自定义 Source 来解决这个问题。自定义 Source 允许根据特定需求编写自己的数据源逻辑，以满足定制化的数据读取需求。

自定义 Source 时，可以通过实现 SourceFunction 接口自定义生成数据的逻辑。SourceFunction 接口定义了两个方法，分别是 run()和 cancel()，其中 run()方法用于自定义生成数据的逻辑并发送生成的数据，在该方法中用户可以通过 while 循环不断地生成数据；cancel()方法用于终止生成数据，在该方法中用户可以设置一个特殊的标记来终止数据的生成。关于自定义 Source 的程序结构如下。

```
public class MySource implements SourceFunction<dataType> {
    @Override
public void run(SourceContext<dataType> sourceContext) throws Exception{
    }
    @Override
```

```
    public void cancel() {
    }
}
```

上述程序结构中，dataType 用于指定生成数据的数据类型。

接下来通过一个案例来演示如何自定义 Source，具体代码如文件 3-6 所示。

文件 3-6　MySource.java

```
1  public class MySource implements
2          SourceFunction<Tuple2<String,Integer>> {
3      private Boolean label = true;
4      @Override
5      public void run(SourceContext<Tuple2<String,Integer>> sourceContext)
6              throws Exception {
7          Random random = new Random();
8          int count = 0;
9          String[] fruits = {"apple","banana","pear","orange","cherry","grape"};
10         while (label){
11             String fruit = fruits[random.nextInt(6)];
12             Integer buyNum = random.nextInt(100);
13             sourceContext.collect(new Tuple2<>(fruit,buyNum));
14             count++;
15             if(count == 10){
16                cancel();
17             }
18          }
19     }
20     @Override
21     public void cancel() {
22         label = false;
23     }
24 }
```

上述代码中，第 3 行代码定义的变量 label 用于标记自定义数据源是否正在运行。第 10～18 行代码通过 while 循环来不断生成元组类型的数据，元组的第一个元素随机从数组 fruits 选取，而第二个元素为 100 以内的随机整数，并且通过 collect()方法发送生成的每条数据。第 15～17 行代码用于判断生成数据的数量是否为 10，当生成数据的数量等于 10 时，调用 cancel()方法终止生成数据。

在 DataStream 程序中，如果需要从自定义 Source 读取数据，只需要在数据源算子 addSource 中实例化自定义 Source 的类即可。下面通过一个案例来演示如何从自定义 Source 读取数据，创建一个名为 ReadMySource 的 DataStream 程序，该程序能够从自定义 Source 读取数据并将其输出到控制台，具体代码如文件 3-7 所示。

文件 3-7　ReadMySource.java

```
1  public class ReadMySource {
2      public static void main(String[] args) throws Exception {
3          StreamExecutionEnvironment executionEnvironment =
4                  StreamExecutionEnvironment.getExecutionEnvironment();
```

```
5          executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
6          DataStream<Tuple2<String, Integer>> mySourceDataStream =
7                  executionEnvironment.addSource(new MySource());
8          mySourceDataStream.print("mySourceDataStream 的数据");
9          executionEnvironment.execute();
10     }
11 }
```

上述代码中，第 6、7 行代码使用数据源算子 addSource 从自定义 Source 读取数据，并将其转换为 DataStream 对象 mySourceDataStream，该对象的数据类型与自定义 Source 生成数据的数据类型一致。

文件 3-7 的运行结果如图 3-16 所示。

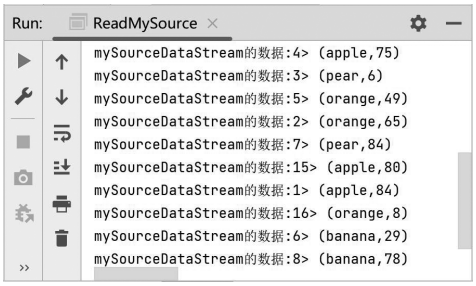

图 3-16　文件 3-7 的运行结果

从图 3-16 可以看出，mySourceDataStream 包含 10 条数据，并且每条数据与自定义 Source 中 run()方法指定生成数据的逻辑一致。因此说明，成功使用了数据源算子 addSource 从自定义 Source 读取数据。

3.5　数据转换

从不同的数据源读取数据之后，便可以使用 DataStream API 提供的算子对 DataStream 对象进行转换，这些算子称为转换算子(Transformation Operator)。转换算子可以将一个或多个 DataStream 对象转换为新的 DataStream 对象，这些转换算子可以实现各种数据处理逻辑，从而让用户可以灵活地操作数据。本节针对常见的转换算子进行介绍。

3.5.1　map

map 用于对 DataStream 对象的每条数据进行转换，形成新的 DataStream 对象。例如，使用 map 对 DataStream 对象进行转换，将每条数据乘以 2，其转换过程如图 3-17 所示。

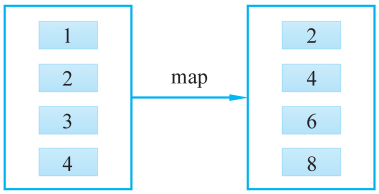

图 3-17　map 转换过程

从图 3-17 可以看出，map 每转换一条数据便输出一条数据作为转换结果。有关使用 map 对 DataStream 对象进行转换的程序结构如下所示。

```
DataStream<OutputDataType> newDataStream = dataStream.map(
    new MapFunction<InputDataType, OutputDataType>() {
    @Override
    public OutputDataType map(InputDataType inputData) throws Exception {
        return resultData;
    }
});
```

上述程序结构中，OutputDataType 用于指定转换结果的数据类型；newDataStream 用于指定新生成 DataStream 对象的名称；dataStream 用于指定转换的 DataStream 对象；InputDataType 用于指定转换的 DataStream 对象的数据类型；inputData 表示转换的 DataStream 对象的每条数据；resultData 用于指定数据的转换结果。

接下来通过一个案例演示如何使用 map 对 DataStream 对象进行转换。创建一个名为 MapDemo 的 DataStream 程序，该程序将集合读取的每条数据乘以 2，具体代码如文件 3-8 所示。

文件 3-8　MapDemo.java

```
1  public class MapDemo {
2     public static void main(String[] args) throws Exception {
3         StreamExecutionEnvironment executionEnvironment =
4                 StreamExecutionEnvironment.getExecutionEnvironment();
5       executionEnvironment.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
6         DataStream<Integer> inputDataStream =
7                 executionEnvironment.fromElements(1, 2, 3);
8         DataStream<Integer> mapDataStream =
9                 inputDataStream.map(new MapFunction<Integer, Integer>() {
10            @Override
11            public Integer map(Integer inputData) throws Exception {
12                Integer resultData = inputData * 2;
13                return resultData;
14            }
15        });
16        mapDataStream.print("mapDataStream 的数据");
17        executionEnvironment.execute();
18    }
19 }
```

上述代码中，第 6、7 行代码使用预定义数据源算子 fromElements 从类型为 Integer 的对象序列读取数据，并将其转换为 DataStream 对象 inputDataStream，该对象的数据类型与对象序列中每个对象的类型一致。第 8～15 行代码使用 map 对 inputDataStream 进行转换，形成新的 DataStream 对象 mapDataStream。

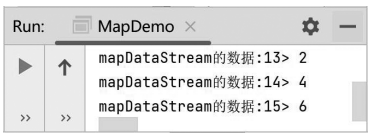

图 3-18　文件 3-8 的运行结果

文件 3-8 的运行结果如图 3-18 所示。

从图 3-18 可以看出，mapDataStream 的数据与 inputDataStream 中每条数据乘以 2 的结果一致。因此说

明，使用 map 成功对 DataStream 对象进行转换。

3.5.2 flatMap

flatMap 用于对 DataStream 对象进行扁平化处理，将 DataStream 对象的每条数据按照特定规则拆分为多条数据，再对拆分后的每条数据进行转换，形成新的 DataStream 对象。例如，使用 flatMap 对 DataStream 对象进行转换，将每条数据通过字符“,”拆分成多条数据，拆分后的每条数据乘以 2，其转换过程如图 3-19 所示。

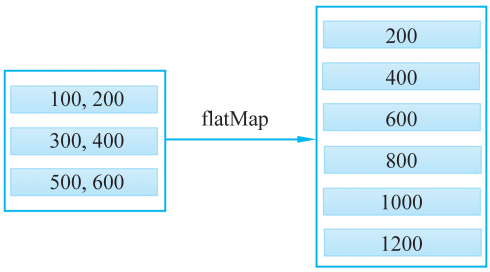

图 3-19 flatMap 转换过程

从图 3-19 可以看出，flatMap 将每条数据拆分为两条数据作为转换结果。有关使用 flatMap 对 DataStream 对象进行转换的程序结构如下所示。

```
DataStream<OutputDataType> newDataStream = dataStream.flatMap(
    new FlatMapFunction<InputDataType, OutputDataType>() {
    @Override
    public void flatMap(
                InputDataType inputData,
                    Collector<OutputDataType> collector) throws Exception {
    collector.collect(resultData);
    }
});
```

上述程序结构中，OutputDataType 用于指定转换结果的数据类型；newDataStream 用于指定新生成 DataStream 对象的名称；dataStream 用于指定转换的 DataStream 对象；InputDataType 用于指定转换的 DataStream 对象的数据类型；inputData 表示转换的 DataStream 对象的每条数据；resultData 用于指定数据的转换结果；collector 用于收集输出的数据，通过调用 collector 提供的 collect()方法，可以将收集的数据输出到新生成的 DataStream 对象。

接下来通过一个案例演示如何使用 flatMap 对 DataStream 对象进行转换。创建一个名为 FlatMapDemo 的 DataStream 程序，该程序将集合读取的每条数据通过分隔符“,”进行拆分，拆分后的每条数据乘以 2，具体代码如文件 3-9 所示。

文件 3-9 FlatMapDemo.java

```
1  public class FlatMapDemo {
2      public static void main(String[] args) throws Exception {
3          StreamExecutionEnvironment executionEnvironment =
4                  StreamExecutionEnvironment.getExecutionEnvironment();
```