实时计算概述和流计算开发

```
学习目标：
1. 掌握实时计算的具体含义，了解Flink的应用场景
2. 掌握Flink的架构体系，了解如何安装配置Flink服务
3. 掌握Flink的基本概念
4. 掌握Flink的数据流处理方式，学会如何开发Flink应用
```

1 Flink概述

1.1 数据流与流计算

数据流是一串连续不断的数据的集合，就象水管里的水流，在水管的一端一点一点地供水，而在水管的另一端看到的是一股连续不断的水流。类似于人们对河流的理解本质上也就是流的概念，但是这条河没有开始也没有结束，数据流非常适合于离散的、没有开头或结尾的数据。例如，交通信号灯的数据是连续的，没有“开始”或“结束”，是连续的过程而不是分批发送的数据记录。通常情况下，数据流对于在生成连续数据流中以小尺寸（通常以KB字节为单位）发送数据的数据源类型是有用的。这包括各种各样的数据源，例如来自连接设备的遥测，客户访问的Web应用时生成的日志文件、电子商务交易或来自社交网络或地理LBS服务的信息等。

传统上，数据是分批移动的，批处理通常同时处理大量数据，具有较长时间的延迟。例如，该复制过程每24小时运行一次。虽然这可以是处理大量数据的有效方法，但它不适用于流式传输的数据，因为数据在处理时已经是旧的内容。

采用数据流是时间序列和随时间检测模式的最佳选择。例如，跟踪Web会话的时间。大多数物联网产生的数据非常适合数据流处理，包括交通传感器，健康传感器，交易日志和活动日志等都是数据流的理想选择。

流数据通常用于实时聚合和关联、过滤或采样。通过数据流，我们可以实时分析数据，并深入了解各种行为，例如统计，服务器活动，设备地理位置或网站点击量等。

数据流整合技术的解决方案

- 金融机构跟踪市场变化，并可根据配置约束（例如达到特定股票价格时出售）调整客户组合的配置。
- 电网监控吞吐量并在达到某些阈值时生成警报。
- 新闻资讯APP从各种平台进行流式传输时，产生的点击记录，实时统计信息数据，以便它可以提供与受众人口相关的文章推荐。
- 电子商务站点以数据流传输点击记录，可以检查数据流中的异常行为，并在点击流显示异常行为时发出安全警报。

数据流带给我们的挑战

数据流是一种功能强大的工具，但在使用流数据源时，有一些常见的挑战。以下的列表显示了要规划数据流的一些事项：

- 可扩展性规划
- 数据持久性规划
- 如何在存储层和处理层中加入容错机制

数据流的管理工具

随着数据流的不断增长，出现了许多合适的大数据流解决方案。我们总结了一个列表，这些都是用于处理流数据的常用工具：

- Apache Kafka

Apache Kafka是一个分布式发布/订阅消息传递系统，它集成了应用程序和数据流处理。

- Apache Storm

Apache Storm是一个分布式实时计算系统。Storm用于分布式机器学习、实时分析处理，尤其是其具有超高数据处理的能力。

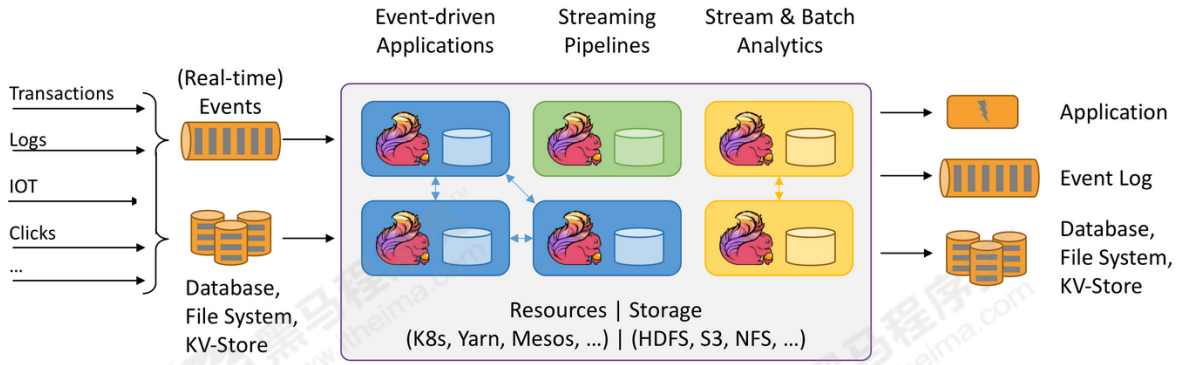
- Apache Flink

Apache Flink是一种数据流引擎，为数据流上的分布式计算提供了诸多便利。

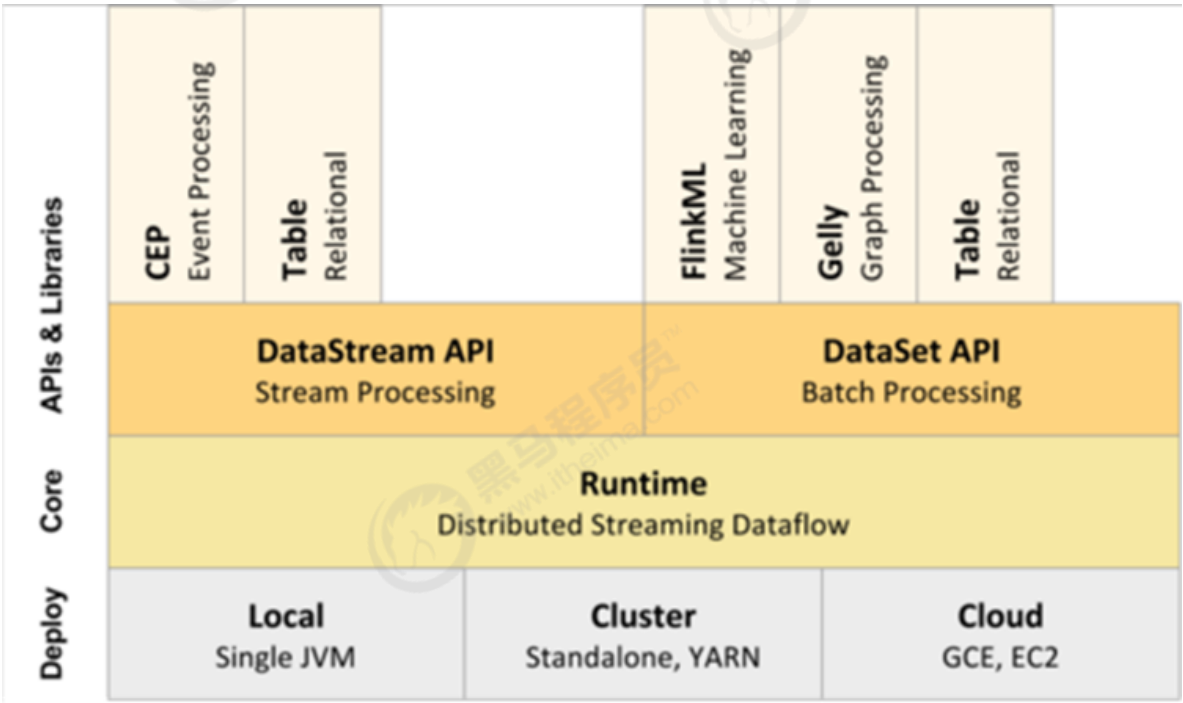
1.2 Flink简介

Apache Flink 是一个开源的分布式流式处理框架，是新的流数据计算引擎，用java实现。Flink可以：

- 提供准确的结果，甚至在出现无序或者延迟加载的数据的情况下。
- 它是状态化的容错的，同时在维护一次完整的的应用状态时，能无缝修复错误。
- 大规模运行，在上千个节点运行时有很好的吞吐量和低延迟。

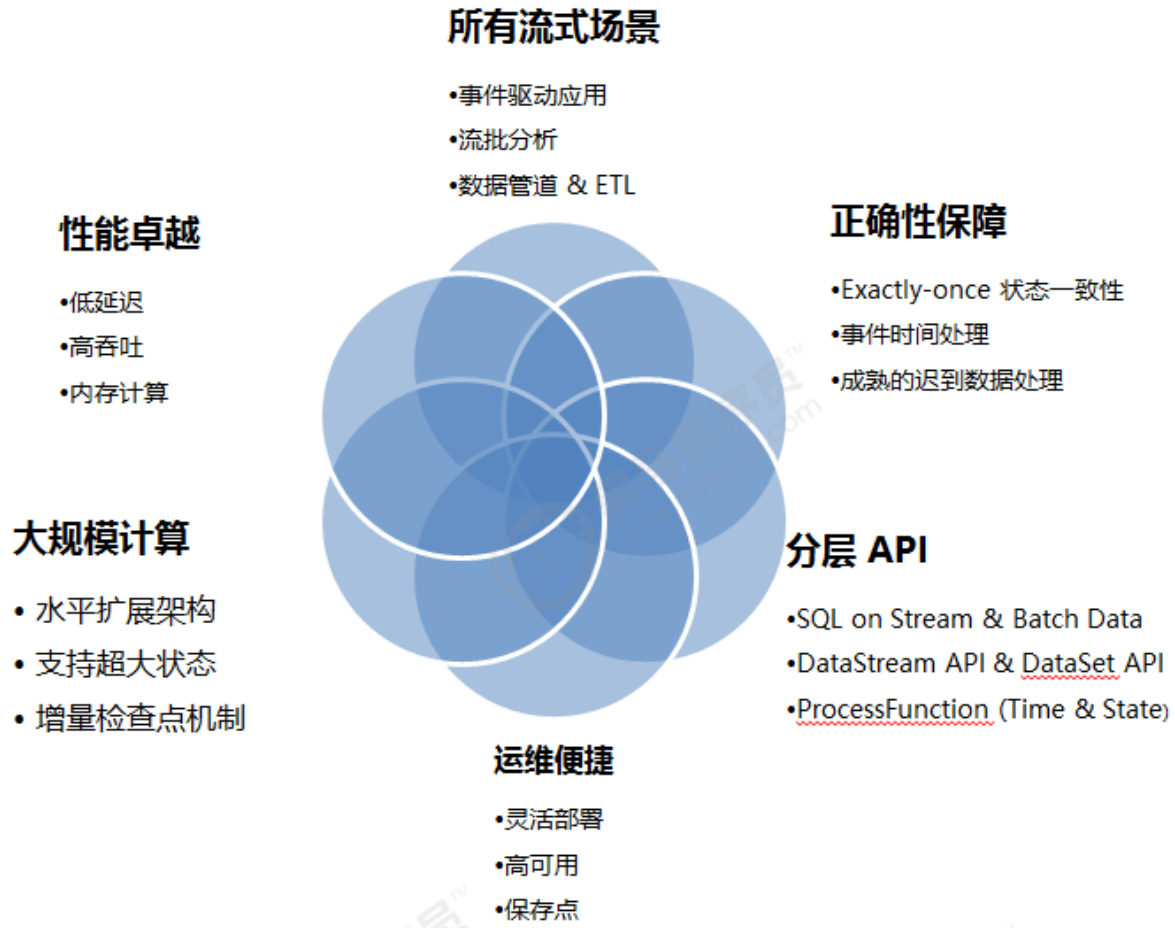


Flink的核心组件：



1.3 应用场景

Apache Flink 功能强大，支持开发和运行多种不同种类的应用程序。它的主要特性包括：



Flink 不仅可以运行在包括 YARN、 Mesos、Kubernetes 在内的多种资源管理框架上，还支持在裸机集群上独立部署。在启用高可用选项的情况下，它不存在单点失效问题。事实证明，Flink 已经可以扩展到数千核心，其状态可以达到 TB 级别，且仍能保持高吞吐、低延迟的特性。世界各地有很多要求严苛的流处理应用都运行在 Flink 之上。

Flink适用的应用场景包括：

1. 事件驱动型应用

- 反欺诈
- 异常检测
- 基于规则的报警
- 业务流程监控
- （社交网络）Web 应用

2. 数据分析应用

- 电信网络质量监控
- 移动应用中的产品更新及实验评估分析
- 消费者技术中的实时数据即席分析
- 大规模图分析

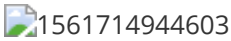
3. 数据管道应用

- 电商中的实时查询索引构建
- 电商中的持续 ETL

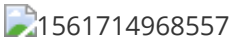
1.4 Flink架构

Flink在运行中主要有三个组件组成，JobClient，JobManager 和 TaskManager。

工作原理如下图 ：

 1561714944603

作业提交流程如下图:

 1561714968557

- Program Code：我们编写的 Flink 应用程序代码。
- Job Client：Job Client 不是 Flink 程序执行的内部部分，但它是任务执行的起点。 Job Client 负责接受用户的程序代码，然后创建数据流，将数据流提交给 Job Manager 以便进一步执行。 执行完成后，Job Client 将结果返回给用户。
- Job Manager：主进程（也称为作业管理器）协调和管理程序的执行。 它的主要职责包括安排任务，管理checkpoint ，故障恢复等。机器集群中至少要有一个 master，master 负责调度 task，协调 checkpoints 和容灾，高可用设置的话可以有多个 master，但要保证一个是 leader, 其他是 standby; Job Manager 包含 Actor system、Scheduler、Check pointing 三个重要的组件。
- Task Manager：从 Job Manager 处接收需要部署的 Task。Task Manager 是在 JVM 中的一个或多个线程中执行任务的工作节点。 任务执行的并行性由每个 Task Manager 上可用的任务槽决定。 每个任务代表分配给任务槽的一组资源。 例如，如果 Task Manager 有四个插槽，那么它将为每个插槽分配 25% 的内存。 可以在任务槽中运行一个或多个线程。 同一插槽中的线程共享相同的 JVM。 同一 JVM 中的任务共享 TCP 连接和心跳消息。Task Manager 的一个 Slot 代表一个可用线程，该线程具有固定的内存，注意 Slot 只对内存隔离，没有对 CPU 隔离。默认情况下，Flink 允许子任务共享 Slot，即使它们是不同 task 的 subtask，只要它们来自相同的 job。这种共享可以有更好的资源利用率。

1.5 安装配置

Flink的运行一般分为三种模式，即local、Standalone、On Yarn。

下载程序：

```
[root@node2-vm06 opt]# cd /opt
[root@node2-vm06 opt]# wget -c
http://mirrors.tuna.tsinghua.edu.cn/apache/flink/flink-1.9.1/flink-1.9.1-bin-
scala_2.12.tgz

[root@node2-vm06 opt]# tar xzf flink-1.9.1-bin-scala_2.12.tgz
```

1. **Local模式**

Local模式比较简单，用于本地测试，安装过程也比较简单，只需在主节点上解压安装包就代表成功安装了，在flink安装目录下使用**./bin/start-cluster.sh**（windows环境下是.bat）命令，就可以通过master:8081监控集群状态，关闭集群命令：**./bin/stop-cluster.sh**（windows环境下是.bat）。

2. **Standalone模式**

Standalone模式顾名思义，是在本地集群上调度执行，不依赖于外部调度机制例如YARN。此时需要对配置文件进行一些简单的修改，我们预计使用当前服务器充当Job manager和Task Manager，一般情况下需要多台机器。

在安装Flink之前，需要对安装环境进行检查，对于Standalone模式，需要提前安装好zookeeper。

1) 修改环境变量，vim /etc/profile,添加以下内容

```
export FLINK_HOME=/opt/flink-1.9.1/
export PATH=$FLINK_HOME/bin:$PATH
```

2) 更改配置文件flink-conf.yaml，

cd /opt/flink-1.9.1/conf

vim flink-conf.yaml

```
# 主要更改的位置有：
jobmanager.rpc.address: 172.17.0.143
taskmanager.numberOfTaskSlots: 2
parallelism.default: 4

#取消下面两行的注释
rest.port: 8081
rest.address: 0.0.0.0
```

上述我们只列出了一些常用需要修改的文件内容，下面我们再简单介绍一些

```
# jobManager 的IP地址
jobmanager.rpc.address:  172.17.0.143

# JobManager 的端口号
jobmanager.rpc.port: 6123

# JobManager JVM heap 内存大小
jobmanager.heap.size: 1024m

# TaskManager JVM heap 内存大小
taskmanager.heap.size: 1024m

# 每个 TaskManager 提供的任务 slots 数量大小，默认为1
taskmanager.numberOfTaskSlots: 2

# 程序默认并行计算的个数，默认为1
parallelism.default: 4
```

2) 配置masters文件

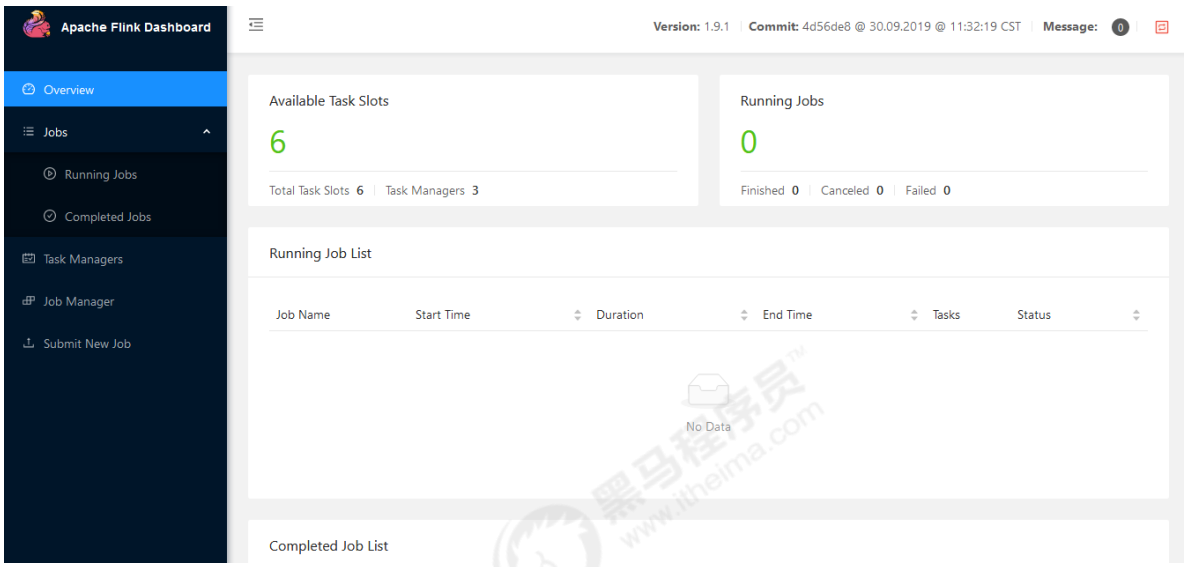
该文件用于指定主节点及其web访问端口，表示集群的jobmanager，**vim masters**，添加 **localhost:8081**

3) 配置slaves文件，该文件用于指定从节点，表示集群的taskManager。添加以下内容

```
localhost
localhost
localhost
```

4) 启动flink集群（因为在环境变量中已经指定了flink的bin位置，因此可以直接输入start-cluster.sh）

5) 验证flink进程，登录web界面，查看Web界面是否正常。至此，standalone模式已成功安装。



1.6 示例演示

Flink安装目录下的example目录里有一些Flink程序示例，我们可以使用这些示例来感受一下Flink的功能。

下面的步骤是我们演示SocketWindowWordCount这个应用的过程，这个应用的作用是监听某个socket服务器端口，实时计算这个端口数据的单词数量。

1. 打开端口

```
# 在nc命令行中输入文本， 必要时需要安装nc命令, yum -y install nc
[root@node2-vm06 streaming]# nc -l 9010
```

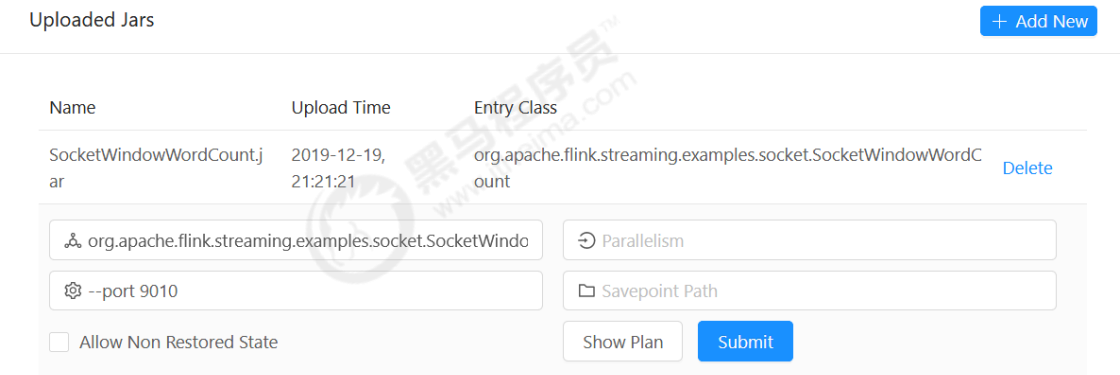
2. 提交示例应用

   有两种方式提交应用：

   1）使用flink命令提交应用

```
[root@node2-vm06 streaming]# /opt/flink-1.9.1/bin/flink run /opt/flink-
1.9.1/examples/streaming/SocketWindowWordCount.jar --port 9010
```

   2）页面上选择应用的jar文件



3. 在nc命令行中输入单词

```
[root@node2-vm06 log]# nc -l 9010
abc abc def dfs def
ttt ttt ggg ggg
```

4. 查看结果

```
def : 2
dfs : 1
abc : 2
ttt : 2
ggg : 2
```

2 单词统计示例

2.1 创建Flink工程

刚才演示的是flink自带的程序，现在我们演示如何通过cmd命令窗口创建flink应用：

```
mvn archetype:generate -DarchetypeGroupId=org.apache.flink -
DarchetypeArtifactId=flink-quickstart-java -DarchetypeVersion=1.9.1
```

依次输入： groupid， artifactId，version，package

 1561715039737

cold-flink工程创建成功，使用idea等工具导入工程，工程目录结构：

```
cold-flink/
├── pom.xml
└── src
    └── main
        ├── java
        |   └── com.itcast.cold.flink
        |       └── BatchJob.java
        |       └── StreamingJob.java
        └── resources
            └── log4j.properties
```

Batchjob： 批处理程序代码

StreamingJob: 流数据程序代码

我们就可以在这两个java类中添加我们的业务逻辑代码了。

2.2 代码实现

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class WindowWordCount {

    public static void main(String[] args) throws Exception {
        //设置运行时环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
```

```
        //设置输入流，并执行数据流的处理和转换
        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .keyBy(0)
                .timeWindow(Time.seconds(5))
                .sum(1);
        //设置输出流
        dataStream.print();

        //执行程序
        env.execute("Window WordCount");
    }

    public static class Splitter implements FlatMapFunction<String,
Tuple2<String, Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>>
out) throws Exception {
            for (String word: sentence.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }

}
```

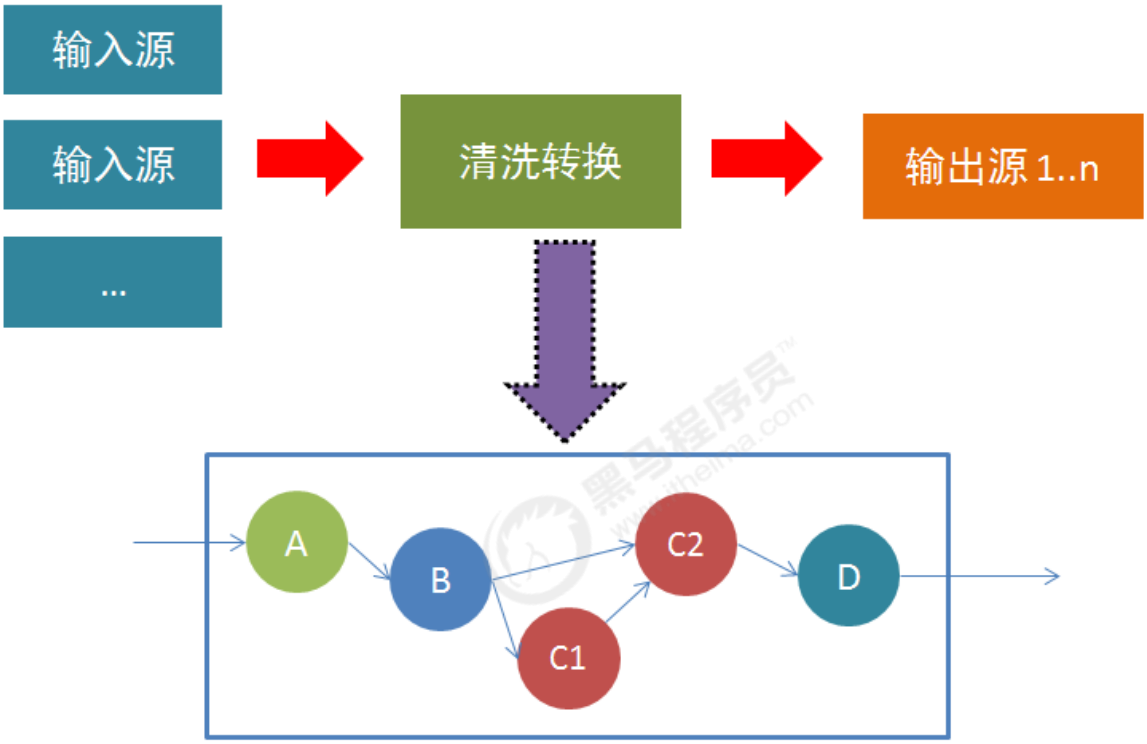
3 基本概念

3.1 DataStream和DataSet

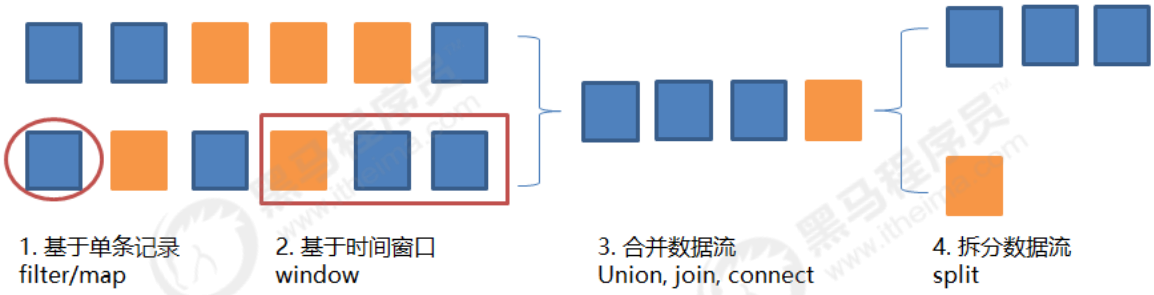
Flink使用DataStream、DataSet在程序中表示数据，我们可以将它们视为可以包含重复项的不可变数据集合。

DataSet是有限数据集（比如某个数据文件），而DataStream的数据可以是无限的（比如kafka队列中的消息）。

这些集合在某些关键方面与常规Java集合不同。首先，它们是不可变的，这意味着一旦创建它们就无法添加或删除元素。你也不能简单地检查里面的元素。



数据流通过filter/map等各种方法，执行过滤、转换、合并、拆分等操作，达到数据计算的目的。



3.2 数据类型

Flink对DataSet或DataStream中可以包含的元素类型设置了一些限制，以便于更有效的执行策略。

有六种不同类别的数据类型：

1. **Java元组和Scala案例类**
2. **Java POJO**
3. **原始类型**
4. **常规类**
5. **值**
6. **Hadoop Writables**

3.2.1 元组

元组是包含固定数量的具有各种类型的字段的复合类型。Java API提供了 `Tuple1` 到 `Tuple25` 。元组的每个字段都可以是包含更多元组的任意Flink类型，从而产生嵌套元组。可以使用字段名称直接访问元组的字段 `tuple.f4` ，或使用通用getter方法 `tuple.getField(int position)` 。字段索引从0开始。请注意，这与Scala元组形成对比，但它与Java的一般索引更为一致。

```
DataStream<Tuple2<String, Integer>> wordCounts = env.fromElements(
    new Tuple2<String, Integer>("hello", 1),
    new Tuple2<String, Integer>("world", 2));

wordCounts.map(new MapFunction<Tuple2<String, Integer>, Integer>() {
    @Override
    public Integer map(Tuple2<String, Integer> value) throws Exception {
        return value.f1;
    }
});

wordCounts.keyBy(0); // also valid .keyBy("f0")
```

3.2.2 POJOs

如果满足以下要求，则Flink将Java和Scala类视为特殊的POJO数据类型：

- 类必须公开类
- 它必须有一个没有参数的公共构造函数（默认构造函数）。
- 所有字段都是公共的，或者必须通过getter和setter函数访问。对于一个名为`foo`的属性的getter和setter方法的字段必须命名`getFoo()`和`setFoo()`。
- 注册的序列化程序必须支持字段的类型。

序列化：

POJO通常使用PojoTypeInfo和PojoSerializer（使用Kryo作为可配置的回退）序列化。例外情况是POJO实际上是Avro类型（Avro特定记录）或生成为“Avro反射类型”。在这种情况下，POJO使用AvroTypeInfo和AvroSerializer序列化。如果需要，您还可以注册自己的自定义序列化程序

```
public class WordWithCount {

    public String word;
    public int count;

    public WordWithCount() {}

    public WordWithCount(String word, int count) {
        this.word = word;
        this.count = count;
    }
}

DataStream<WordWithCount> wordCounts = env.fromElements(
    new WordWithCount("hello", 1),
    new WordWithCount("world", 2));

wordCounts.keyBy("word"); // key by field expression "word"
```

3.2.3 基础数据类型

Flink支持所有Java和Scala的原始类型，如`Integer`，`String`和`Double`。

3.2.4 常规类

Flink支持大多数Java和Scala类（API和自定义）。限制适用于包含无法序列化的字段的类，如文件指针，I / O流或其他本机资源。遵循Java Beans约定的类通常可以很好地工作。

所有未标识为POJO类型的类都由Flink作为常规类类型处理。Flink将这些数据类型视为黑盒子，并且无法访问其内容（例如，用于有效排序）。使用序列化框架Kryo对常规类型进行反序列化。

3.2.5 值

*值*类型手动描述其序列化和反序列化。它们不是通过通用序列化框架，而是通过`org.apache.flinktypes.Value`使用方法`read`和实现接口为这些操作提供自定义代码`write`。当通用序列化效率非常低时，使用值类型是合理的。一个示例是将元素的稀疏向量实现为数组的数据类型。知道数组大部分为零，可以对非零元素使用特殊编码，而通用序列化只需编写所有数组元素。

该`org.apache.flinktypes.CopyableValue`接口以类似的方式支持手动内部克隆逻辑。

Flink带有与基本数据类型对应的预定义值类型。（`ByteValue`，`ShortValue`，`IntValue`，`LongValue`，`FloatValue`，`DoubleValue`，`StringValue`，`CharValue`，`BooleanValue`）。这些值类型充当基本数据类型的可变变体：它们的值可以被更改，允许程序员重用对象并从垃圾收集器中消除压力。

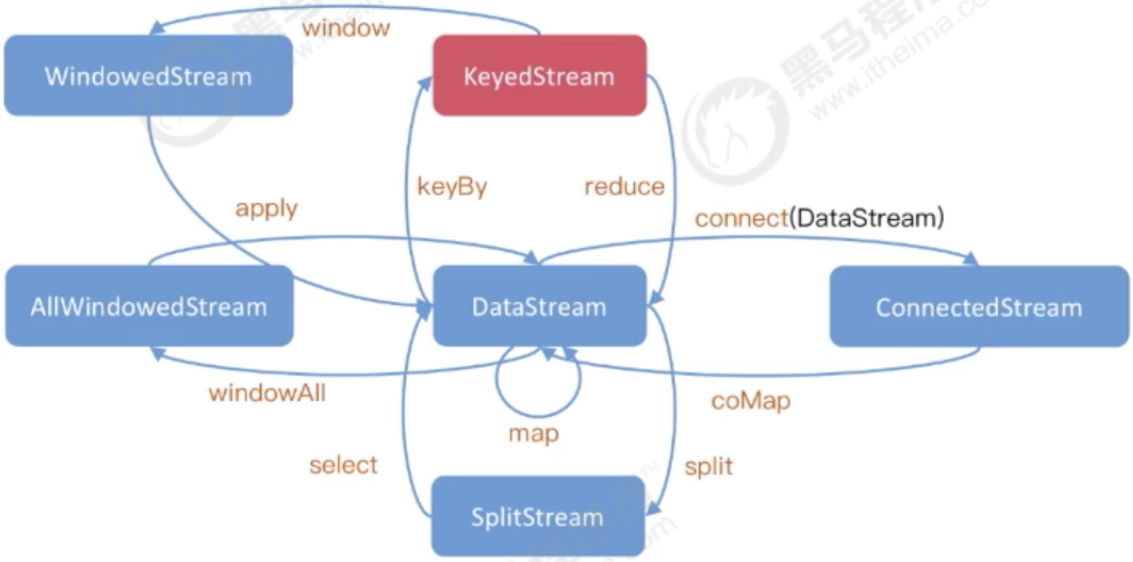
3.2.6. Hadoop Writables

使用实现`org.apache.hadoop.Writable`接口的类型。

`write()`和`readFields()`方法中定义的序列化逻辑将用于序列化。

3.3 数据的操作

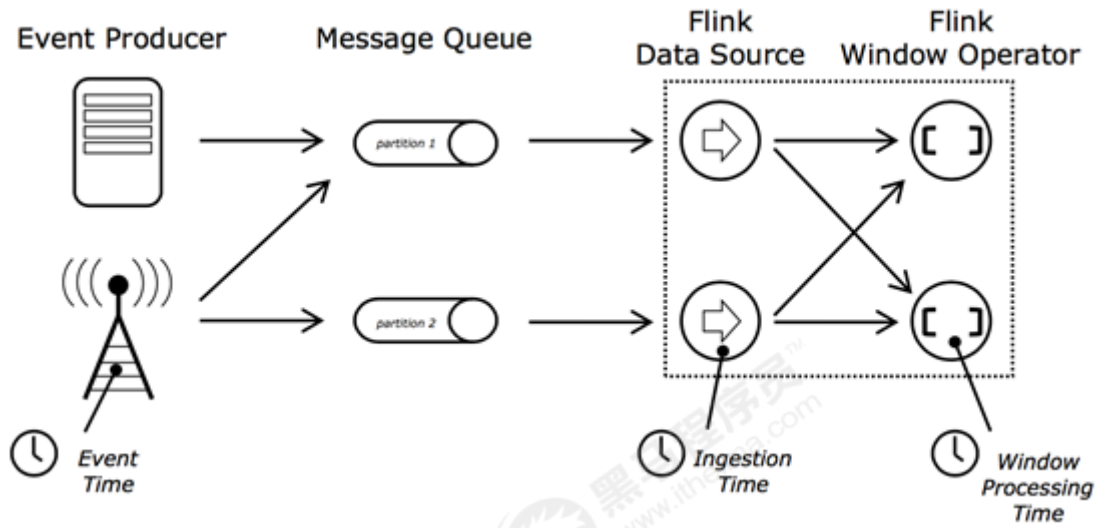
数据转换，即通过从一个或多个 DataStream 生成新的DataStream 的过程，是主要的数据处理的手段。Flink 提供了多种数据转换操作，基本可以满足所有的日常使用场景。



3.4 窗口的含义

Flink计算引擎中，时间是一个非常重要的概念，Flink的时间分为三种时间：

- EventTime: 事件发生的时间
- IngestionTime:事件进入 Flink 的时间
- ProcessingTime:事件被处理时的时间

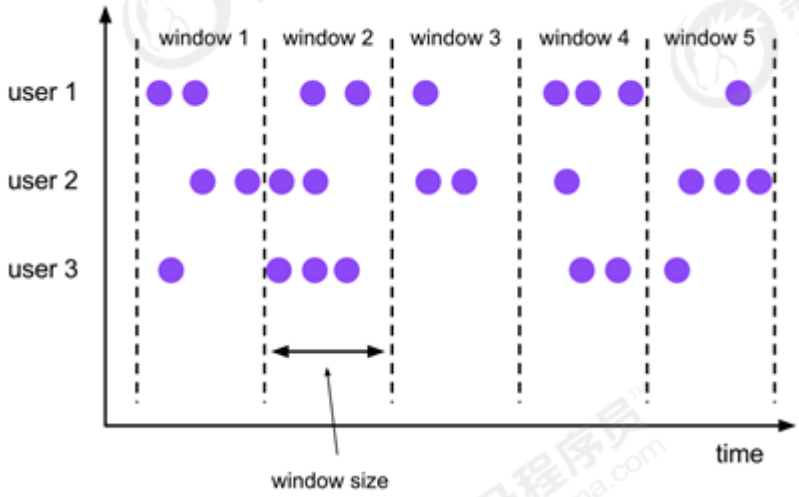


窗口是Flink流计算的一个核心概念，Flink窗口主要包括：

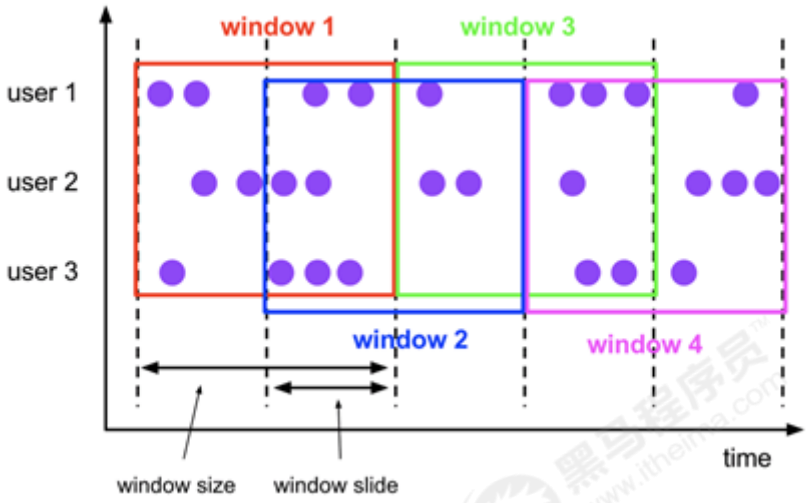
- 时间窗口
  - 翻滚时间窗口
  - 滑动时间窗口
- 数量窗口
  - 翻滚数量窗口
  - 滑动数量窗口

按照形式来划分，窗口又分为：

- 翻滚窗口



- 滑动窗口

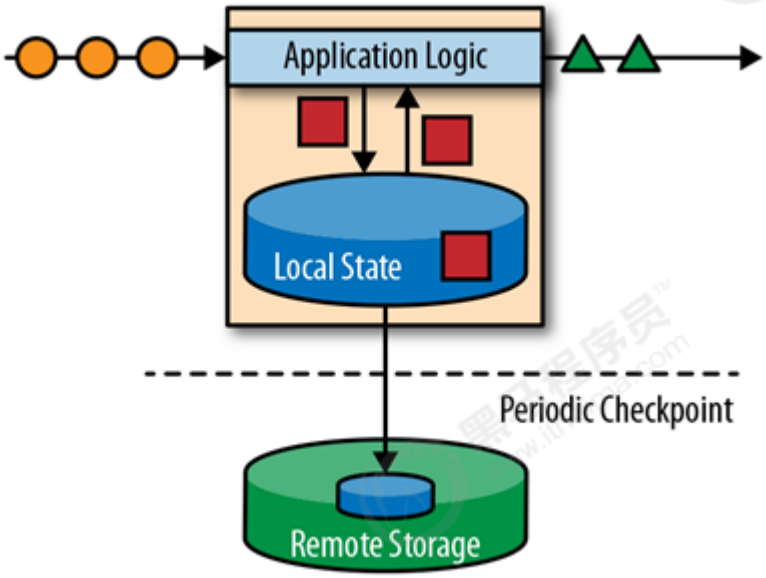


3.5 有状态的流式处理

在很多场景下，数据都是以持续不断的流事件创建。例如网站的交互、或手机传输的信息、服务器日志、传感器信息等。有状态的流处理（stateful stream processing）是一种应用设计模式，用于处理无边界的流事件。

对于任何处理流事件的应用来说，并不会仅仅简单的一次处理一个记录就完事了。在对数据进行处理或转换时，操作应该是有状态的。也就是说，需要有能力做到对处理记录过程中生成的中间数据进行存储及访问。当一个应用收到一个 事件，在对其做处理时，它可以从状态信息（state）中读取数据进行协助处理。或是将数据写入state。在这种准则下，状态信息（state）可以被存储（及访问）在很多不同的地方，例如程序变量，本地文件，或是内置的（或外部的）数据库中。

Apache Flink 存储应用状态信息在本地内存或是一个外部数据库中。因为Flink 是一个分布式系统，本地状态信息需要被有效的保护，以防止在应用或是硬件挂掉之后，造成数据丢失。Flink对此采取的机制是：定期为应用状态（application state）生成一个一致（consistent）的checkpoint，并写入到一个远端持久性的存储中。下面是一个有状态的流处理Flink application的示例图：



Stateful stream processing 应用的输入一般为：事件日志（event log）的持续事件。Event log 存储并且分发事件流。事件被写入一个持久性的，仅可追加的（append-only）日志中。也就是说，被写入的事件的顺序始终是不变的。所以事件在发布给多个不同用户时，均是以完全一样的顺序发布的。在开源的event log 系统中，最著名的当属 Kafka。

使用flink流处理程序连接event log的理由有多种。在这个架构下，event log 持久化输入的 events，并且可以以既定的顺序重现这些事件。万一应用发生了某个错误，Flink会通过前一个checkpoint 恢复应用的状态，并重置在event log 中的读取位置，并据此对events做重现，直到它抵达stream 的末端。这个技术不仅被用于错误恢复，并且也可以用于更新应用，修复bugs，以及修复之前遗漏结果等场景中。

3 DataStream编程

3.1 输入流

源是程序从中读取输入的位置，可以使用以下方法将源附加到您的程序：`StreamExecutionEnvironment.addSource(sourceFunction)`。

Flink附带了许多预先实现的源函数，但您可以通过实现`SourceFunction`非并行源，或通过实现`ParallelSourceFunction`接口或扩展`RichParallelSourceFunction`for parallel源来编写自己的自定义源。

有几个预定义的流源可从以下位置访问`StreamExecutionEnvironment`：

基于文件：

- `readTextFile(path)` - `TextInputFormat`逐行读取文本文件，即符合规范的文件，并将它们作为字符串返回。
- `readFile(fileInputFormat, path)` - 按指定的文件输入格式指定读取（一次）文件。
- `readFile(fileInputFormat, path, watchType, interval, pathFilter, typeInfo)` - 这是前两个内部调用的方法。它`path`根据给定的内容读取文件`fileInputFormat`。根据提供的内容`watchType`，此源可以定期监视（每`interval`ms）新数据（`FileProcessingMode.PROCESS_CONTINUOUSLY`）的路径，或者处理当前在路径中的数据并退出（`FileProcessingMode.PROCESS_ONCE`）。使用`pathFilter`，用户可以进一步排除正在处理的文件。

  实现：

  Flink将文件读取过程分为两个子任务，即*目录监控*和*数据读取*。这些子任务中的每一个都由单独的实体实现。监视由单个**非并行**（并行性= 1）任务实现，而读取由并行运行的多个任务执行。后者的并行性等于工作并行性。单个监视任务的作用是扫描目录（定期或仅一次，具体取决于`watchType`），找到要处理的文件，将它们*分成分割*，并将这些拆分分配给下游读者。读者是那些将阅读实际数据的人。每个分割仅由一个读取器读取，而读取器可以逐个读取多个分割。

  重要笔记：

  1. 如果`watchType`设置为`FileProcessingMode.PROCESS_CONTINUOUSLY`，则在修改文件时，将完全重新处理其内容。这可以打破“完全一次”的语义，因为在文件末尾附加数据将导致其**所有**内容被重新处理。
  2. 如果`watchType`设置为`FileProcessingMode.PROCESS_ONCE`，则源扫描路径**一次**并退出，而不等待读者完成读取文件内容。当然读者将继续阅读，直到读取所有文件内容。在该点之后关闭源将导致不再有检查点。这可能会导致节点故障后恢复速度变慢，因为作业将从上一个检查点恢复读取。

基于Socket：

- `socketTextStream` - 从Socket中读取，元素可以用分隔符分隔。

基于集合:

- `fromCollection(Collection)` - 从Java Java.util.Collection创建数据流。集合中的所有元素必须属于同一类型。

- `fromCollection(Iterator, Class)` - 从迭代器创建数据流。该类指定迭代器返回的元素的数据类型。
- `fromElements(T ...)` - 从给定的对象序列创建数据流。所有对象必须属于同一类型。
- `fromParallelCollection(SplittableIterator, Class)` - 并行地从迭代器创建数据流。该类指定迭代器返回的元素的数据类型。
- `generateSequence(from, to)` - 并行生成给定间隔中的数字序列。

自定义：

- `addSource` - 附加新的源功能。例如，要从Apache Kafka读取，可以使用 `addSource(new FlinkKafkaConsumer08<>(...))`。

3.2 数据流转换

运算符将一个或多个DataStream转换为新的DataStream。程序可以将多个转换组合成复杂的数据流拓扑。

 1564826943736

1. **Map**： DataStream → DataStream:

   调用用户定义的MapFunction对DataStream[T]数据进行处理，形成新的DataStream[T]，其中**数据格式可能会发生变化**,常用作对数据集内数据的清洗和转换。

   如以下示例：它将输入流的元素数值增加一倍：

   ```
   DataStream<Integer> dataStream = //...
   dataStream.map(new MapFunction<Integer, Integer>() {
       @Override
       public Integer map(Integer value) throws Exception {
           return 2 * value;
       }
   });
   ```

2. **FlatMap**：DataStream → DataStream

   主要对输入的元素处理之后生成一个或者多个元素，如下示例：将句子拆分成单词：

   ```
   dataStream.flatMap(new FlatMapFunction<String, String>() {
       @Override
       public void flatMap(String value, Collector<String> out)
           throws Exception {
           for(String word: value.split(" ")){
               out.collect(word);
           }
       }
   });
   ```

3. **Filter**： DataStream → DataStream

   该算子将按照条件对输入数据集进行筛选操作，将符合条件的数据集输出，将不符合条件的数据过滤掉。

   如下所示：返回不为0的数据

```
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }
});
```

4. **KeyBy：** DataStream → KeyedStream

   该算子根据指定的key将输入的DataStream[T]数据格式转换为KeyedStream[T]，也就是在数据集中执行Partition操作，将相同的key值的数据放置在相同的分区中。简单来说，就是sql里面的group by

   ```
   dataStream.keyBy("someKey") // Key by field "someKey"
   dataStream.keyBy(0) // Key by the first element of a Tuple
   ```

   注意 如果出现以下情况，则类型**不能成为Key**：

   1. 它是POJO类型，但不覆盖*hashCode（）*方法并依赖于*Object.hashCode（）*实现。
   2. 它是任何类型的数组。

5. **Reduce：** KeyedStream → DataStream

   该算子和MapReduce的Reduce原理基本一致，主要目的是将输入的KeyedStream通过传入的用户自定义的ReduceFunction滚动的进行数据聚合处理，其中定义的ReduceFunction必须满足运算结合律和交换律：

   ```
   keyedStream.reduce(new ReduceFunction<Integer>() {
       @Override
       public Integer reduce(Integer value1, Integer value2)
       throws Exception {
           return value1 + value2;
       }
   });
   ```

6. **Fold：** KeyedStream → DataStream

   具有初始值的键控数据流上的“滚动”折叠。将当前元素与最后折叠的值组合并发出新值。

   折叠函数，当应用于序列（1,2,3,4,5）时，发出序列“start-1”，“start-1-2”，“start-1-2-3”,. ..

   ```
   DataStream<String> result =
     keyedStream.fold("start", new FoldFunction<Integer, String>() {
       @Override
       public String fold(String current, Integer value) {
           return current + "-" + value;
       }
     });
   ```

7. **Aggregations：** KeyedStream → DataStream

   滚动聚合数据流上的聚合。min和minBy之间的差异是min返回最小值，而minBy返回该字段中具有最小值的元素（max和maxBy相同）。

```
keyedStream.sum(0);
keyedStream.sum("key");
keyedStream.min(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.max("key");
keyedStream.minBy(0);
keyedStream.minBy("key");
keyedStream.maxBy(0);
keyedStream.maxBy("key");
```

8. **Window** KeyedStream → WindowedStream

   可以在已经分区的KeyedStream上定义时间窗口。

   时间窗口根据某些特征（例如，在最后5秒内到达的数据）对每个Key中的数据进行分组。

```
// 最后5秒的数据
dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5)));
```

9. **WindowAll**：DataStream → AllWindowedStream

   Windows可以在常规DataStream上定义。Windows根据某些特征（例如，在最后5秒内到达的数据）对所有流事件进行分组。

   警告：在许多情况下，这**是非并行**转换。所有记录将收集在windowAll运算符的一个任务中。

```
// 最后5秒的数据
    dataStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)));
```

10. **Window Apply** WindowedStream → DataStream AllWindowedStream → DataStream

    将一般功能应用于整个窗口。下面是一个手动求和窗口元素的函数。

    注意：如果正在使用windowAll转换，则需要使用AllWindowFunction。

```
windowedStream.apply (new WindowFunction<Tuple2<String,Integer>, Integer,
Tuple, Window>() {
    public void apply (Tuple tuple,
            Window window,
            Iterable<Tuple2<String, Integer>> values,
            Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});

// applying an AllWindowFunction on non-keyed window stream
allWindowedStream.apply (new AllWindowFunction<Tuple2<String,Integer>,
Integer, Window>() {
    public void apply (Window window,
            Iterable<Tuple2<String, Integer>> values,
            Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
```

```
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});
```

11. **Window Reduce** WindowedStream → DataStream

    将减少功能应用于窗口并返回减少的值。

```
windowedStream.reduce (new ReduceFunction<Tuple2<String,Integer>>() {
    public Tuple2<String, Integer> reduce(Tuple2<String, Integer> value1,
Tuple2<String, Integer> value2) throws Exception {
        return new Tuple2<String,Integer>(value1.f0, value1.f1 + value2.f1);
    }
});
```

12. **Window Fold**：WindowedStream → DataStream

    将折叠功能应用于窗口并返回折叠值。

    示例函数应用于序列（1,2,3,4,5）时，将序列折叠为字符串“start-1-2-3-4-5”：

```
windowedStream.fold("start", new FoldFunction<Integer, String>() {
    public String fold(String current, Integer value) {
        return current + "-" + value;
    }
});
```

13. **Windows上的聚合** WindowedStream → DataStream

    聚合窗口的内容。min和minBy之间的差异是min返回最小值，而minBy返回该字段中具有最小值的元素（max和maxBy相同）。

```
windowedStream.sum(0);
windowedStream.sum("key");
windowedStream.min(0);
windowedStream.min("key");
windowedStream.max(0);
windowedStream.max("key");
windowedStream.minBy(0);
windowedStream.minBy("key");
windowedStream.maxBy(0);
windowedStream.maxBy("key");
```

14. **Union：** DataStream → DataStream

    将两个或者多个输入的数据集合并成一个数据集，需要保证两个数据集的格式一致，输出的数据集的格式和输入的数据集格式保持一致

    注意：如果将数据流与其自身联合，则会在结果流中获取两次元素。

```
dataStream.union(otherStream1, otherStream2, ...);
```

15. **Window Join：**DataStream,DataStream → DataStream

    根据主键和公共时间窗口，连接数据流

```
dataStream.join(otherStream)
    .where(<key selector>).equalTo(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.seconds(3)))
    .apply (new JoinFunction () {...});
```

16. **Interval Join：** KeyedStream，KeyedStream → DataStream

    在给定的时间间隔内使用公共Key连接两个键控流的两个元素e1和e2，以便e1.timestamp + lowerBound <= e2.timestamp <= e1.timestamp + upperBound

```
// this will join the two streams so that
// key1 == key2 && leftTs - 2 < rightTs < leftTs + 2
keyedStream.intervalJoin(otherKeyedStream)
    .between(Time.milliseconds(-2), Time.milliseconds(2)) // lower and upper
bound
    .upperBoundExclusive(true) // optional
    .lowerBoundExclusive(true) // optional
    .process(new IntervalJoinFunction() {...});
```

17. **Window CoGroup：** DataStream，DataStream → DataStream

    在给定Key和公共时间窗口上对两个数据流进行coGroup操作。

```
dataStream.coGroup(otherStream)
    .where(0).equalTo(1)
    .window(TumblingEventTimeWindows.of(Time.seconds(3)))
    .apply (new CoGroupFunction () {...});
```

18. **Connect：** DataStream,DataStream → ConnectedStreams

    Connect算子主要是为了合并两种或者多种不同数据类型的数据集，合并后会保留原来的数据集的数据类型。连接操作允许共享状态数据，也就是说在多个数据集之间可以操作和查看对方数据集的状态。

```
DataStream<Integer> someStream = //...
DataStream<String> otherStream = //...

ConnectedStreams<Integer, String> connectedStreams =
someStream.connect(otherStream);
```

19. **CoMap，CoFlatMap：** ConnectedStreams → DataStream

    类似于连接数据流上的map和flatMap

```
connectedStreams.map(new CoMapFunction<Integer, String, Boolean>() {
    @Override
    public Boolean map1(Integer value) {
        return true;
    }

    @Override
    public Boolean map2(String value) {
        return false;
    }
});
connectedStreams.flatMap(new CoFlatMapFunction<Integer, String, String>() {
```

```
    @Override
    public void flatMap1(Integer value, Collector<String> out) {
        out.collect(value.toString());
    }

    @Override
    public void flatMap2(String value, Collector<String> out) {
        for (String word: value.split(" ")) {
          out.collect(word);
        }
    }
});
```

20. **Split**： DataStream → SplitStream

    根据某些标准将流拆分为两个或更多个流。

```
SplitStream<Integer> split = someDataStream.split(new
OutputSelector<Integer>() {
    @Override
    public Iterable<String> select(Integer value) {
        List<String> output = new ArrayList<String>();
        if (value % 2 == 0) {
            output.add("even");
        }
        else {
            output.add("odd");
        }
        return output;
    }
});
```

21. **Select：** SplitStream → DataStream

    从拆分流中选择一个或多个流。

```
SplitStream<Integer> split;
DataStream<Integer> even = split.select("even");
DataStream<Integer> odd = split.select("odd");
DataStream<Integer> all = split.select("even","odd");
```

22. **Iterate**： DataStream → IterativeStream → DataStream

    通过将一个运算符的输出重定向到某个先前的运算符，在流中创建“反馈”循环。这对于定义不断更新模型的算法特别有用。以下代码以流开头并连续应用迭代体。大于0的元素将被发送回反馈通道，其余元素将向下游转发。

```
IterativeStream<Long> iteration = initialStream.iterate();
DataStream<Long> iterationBody = iteration.map (/*do something*/);
DataStream<Long> feedback = iterationBody.filter(new FilterFunction<Long>(){
    @Override
    public boolean filter(Long value) throws Exception {
        return value > 0;
    }
});
iteration.closeWith(feedback);
```

```
DataStream<Long> output = iterationBody.filter(new FilterFunction<Long>(){
    @Override
    public boolean filter(Long value) throws Exception {
        return value <= 0;
    }
});
```

23. **提取时间戳：** DataStream → DataStream

    从记录中提取时间戳，以便使用事件时间语义的窗口。

    ```
    stream.assignTimestamps (new TimeStampExtractor() {...});
    ```

24. 元组数据流转换：

    Project： DataStream→DataStream

    从元组中选择字段的子集

    ```
    DataStream<Tuple3<Integer, Double, String>> in = // [...]
    DataStream<Tuple2<String, Integer>> out = in.project(2,0);
    ```

3.3 输出流

数据接收器使用DataStream并将它们转发到文件，套接字，外部系统或打印它们。Flink带有各种内置输出格式，这些格式封装在DataStreams上的操作后面：

- `writeAsText()` / `TextOutputFormat` - 按字符串顺序写入元素。通过调用每个元素的*toString（）*方法获得字符串。
- `writeAsCsv(...)` / `CsvOutputFormat` - 将元组写为逗号分隔值文件。行和字段分隔符是可配置的。每个字段的值来自对象的*toString（）*方法。
- `print()` / `printToErr()` - 在标准输出/标准错误流上打印每个元素的*toString（）*值。可选地，可以提供前缀（msg），其前缀为输出。这有助于区分不同的*打印*调用。如果并行度大于1，则输出也将以生成输出的任务的标识符为前缀。
- `writeUsingOutputFormat()` / `FileOutputFormat` - 自定义文件输出的方法和基类。支持自定义对象到字节的转换。
- `writeToSocket` - 根据a将元素写入套接字 `SerializationSchema`
- `addSink` - 调用自定义接收器功能。Flink捆绑了其他系统（如Apache Kafka）的连接器，这些系统实现为接收器功能。

请注意，`write*()` 方法 `DataStream` 主要用于调试目的。他们没有参与Flink的检查点，这意味着这些函数通常具有至少一次的语义。刷新到目标系统的数据取决于OutputFormat的实现。这意味着并非所有发送到OutputFormat的元素都会立即显示在目标系统中。此外，在失败的情况下，这些记录可能会丢失。

为了可靠，准确地将流传送到文件系统，请使用 `flink-connector-filesystem`。此外，通过该 `.addSink(...)` 方法的自定义实现可以参与Flink的精确一次语义检查点。

3.4 示例讲解

4 本章总结

本章介绍了flink流处理框架的应用场景、架构设计以及安装、示例等，同时也介绍了Flink体系的基本概念和对于流数据的操作，包括flink处理流数据时需要经过输入源、数据流转换、输出源等步骤。

程序结构:

```
//1. 创建执行环境
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.createLocalEnvironment()
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.createRemoteEnvironment(host: String, port: Int,
jarFiles: String*)

//2. 添加数据源
//2.1 基于文件
readTextFile(path) //逐行读取文本文件，即符合TextInputFormat规范的文件，并将其作为字符串返回。
readFile(fileInputFormat, path) //按指定的文件输入格式指定读取（一次）文件。
readFile(fileInputFormat, path, watchType, interval, pathFilter) //这是前两个方法在内部调用的方法。

//2.2 基于Socket
socketTextStream(host, port) //监听socket端口数据,按字符格式返回

//2.3 基于集合
fromCollection(Seq) //从Java.util.Collection创建数据流。集合中的所有元素必须是相同类型的。
fromCollection(Iterator) //从迭代器创建数据流。该类指定迭代器返回的元素的数据类型。
fromElements(elements:_*) //给定的对象序列中创建数据流。所有对象必须具有相同的类型。
fromParallelCollection(SplittableIterator)//并行从迭代器创建数据流,该类指定迭代器返回元素的数据类型。
generateSequence(from,to) //在给定的区间内并行生成数字序列。
//2.4 自定义
StreamExecutionEnvironment.addSource(sourceFunction)
DataStream<WikipediaEditEvent> edits = see.addSource(new
WikipediaEditsSource());

//3. 转换
datastream.map(new MapFunction<String, Integer>() { //DataStream --map--
>DataStream
            @Override
            public Integer map(String s) throws Exception {
                return Integer.valueOf(s);
            }
        }).flatMap(new FlatMapFunction<Integer, String>() { //DataStream--
flatMap--->DataStream
            @Override
            public void flatMap(Integer integer, Collector<String> collector)
throws Exception {
                collector.collect(String.valueOf(integer));
            }
        })
        .keyBy("key1") //DataStream --keyBy-->KeyedStream
                .timeWindow(Time.seconds(5))
                .reduce(new ReduceFunction<String>() { //keyedStream --reduce--
>DataStream
```

```
                    @Override
                    public String reduce(String s, String t1) throws Exception {
                        return s + t1;
                    }
                })
//4. 输出源
//按字符串顺序写入元素。
//通过调用每个元素的toString()方法获得字符串。
writeAsText()

//将元组写为逗号分隔值文件,行和字段分隔符是可配置的.
//每个字段的值来自对象的toString()方法。
writeAsCsv(...)

//在标准输出/标准错误流上打印每个元素的toString()值。
//还可以选择在输出之前提供前缀(msg)。这可以帮助区分不同的打印调用。
//如果并行度大于1，则输出还将加上生成输出的任务的标识符。
print()

//自定义文件输出的方法和基类。
//支持自定义对象到字节的转换。
writeUsingOutputFormat()

//根据SerializationSchema将元素写入Socket
writeToSocket

//调用自定义接收器功能。
//Flink捆绑了其他系统（如Apache Kafka）的连接器，这些系统实现为接收器功能。
addSink
```