

Table of Contents

0. 前言	1.1
1. 状态化流处理概述	1.2
2. 流处理基础	1.3
3. Apache Flink 架构	1.4
4. 设置 Apache Flink 开发环境	1.5
5. DataStream API	1.6
6. 基于时间和窗口的算子	1.7
7. 有状态算子和应用	1.8
8. 读写外部系统	1.9
9. 搭建 Flink 运行流式应用	1.10
10. Flink 和流式应用运维	1.11
11. 还有什么？	1.12

基于 **Apache Flink** 的流处理

本文档为书籍笔记，适合复习查阅使用，不适合初次学习。

- 英文版: [Stream Processing with Apache Flink](#)
- 中文版: [基于 Apache Flink 的流处理](#)

1. 状态化流处理概述

传统数据处理框架

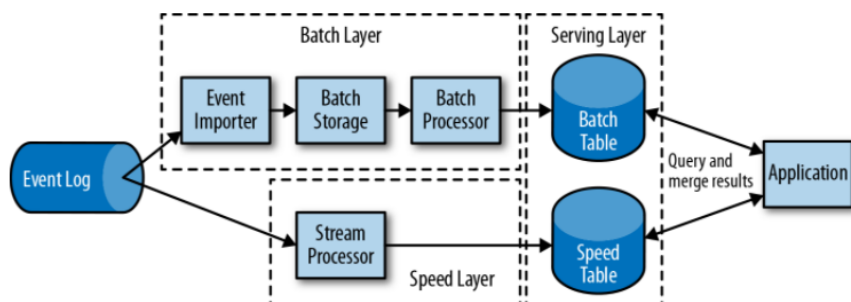
- 事务型处理
 - 数据存储层+数据处理层
 - e.g. ERP、CRM、web应用
- 分析型处理
 - 定期报告查询
 - 即席查询（ad-hoc）

状态化流处理

- 事件驱动型应用：实时推荐、模式识别、异常检测
- 数据管道
- 流式分析

开源流处理历史

- Lambda 架构：提速层（提速表）
- 故障处理机制
- 事件时间和顺序依赖



2. 流处理基础

Dataflow 图

- 算子：数据源、数据汇
- Dataflow 图：逻辑图、物理图
- 并行性：数据并行、任务并行
- 数据交换策略：转发、广播、基于键值、随机

并行流处理

- 延迟↓，吞吐↑
- 数据流上的操作
 - 数据接入、数据输出（TCP、文件、Kafka、传感器）
 - 转换操作
 - 滚动聚合
 - 窗口操作：滚动窗口、滑动窗口、会话窗口

时间语义

- 处理时间、事件时间
- 水位线：迟到事件

状态和一致性模型

- 至多一次：没有保障
- 至少一次：持久化事件日志、记录确认
- 精确一次：数据重放机制、内部状态一致性
- 端到端的精确一次

3. Apache Flink 架构

系统架构

- 架构概述
 - 核心功能：分布式数据流处理
 - 集群管理器：Apache Mesos、YARN、Kubernetes
 - 分布式存储：HDFS、S3
 - HA领导选举：ZooKeeper
- 搭建组件
 - JobManager：单个应用程序的执行，JobGraph -> ExecutionGraph -> 申请 slot -> 检查点
 - ResourceManager：管理 TaskManager 中的 slots，管理申请和使用完的释放
 - TaskManager：工作进程包含多个线程，slots 作为资源进行分配
 - Dispatcher：接收提交的应用，启动 JobManager 对应这个应用

.png>)

- 应用部署
 - 框架模式：打包成 JAR 文件，通过客户端提交到运行的服务上
 - 库模式：绑定到应用所在的容器镜像中，常用于微服务架构
- 任务执行

.png>)

- 高可用性设置
 - TaskManager 故障：JobManager 向 ResourceManager 申请更多的 slot
 - JobManager 故障：ZK 请求获取信息，向 ResourceManager 申请 slot，重启重置检查点

Flink 中的数据传输

- TaskManager之间会有一个或者多个TCP连接
- 基于信用值的流量控制：缓冲合并发送数据
- 任务链接：降低本地通信开销（要求相同并行度）

事件时间处理

- 时间戳+水位线
 - 在数据源完成: `SourceFunction`
 - 周期分配器: `AssignerWithPeriodicWatermarks`
 - 定点分配器: `AssignerWithPunctuatedWatermarks`

.png>)

状态管理

- 算子状态 (**operator state**): 列表状态、联合列表状态、广播状态
- 键值分区状态 (**keyed state**): 单值状态、列表状态、映射状态

检查点、保存点及状态恢复

- 一致性检查点: 等待任务处理完所有的输入数据
- Flink 检查点: 检查点分隔符 (**Chandy-Lamport** 分布式快照算法)
- 保存点

4. 设置 Apache Flink 开发环境

在 IDE 中导入书中示例

```
> git clone https://github.com/streaming-with-flink/examples-sca
```

创建 Flink Maven 项目

```
mvn archetype:generate \  
  -DarchetypeGroupId=org.apache.flink \  
  -DarchetypeArtifactId=flink-quickstart-java \  
  -DarchetypeCatalog=https://repository.apache.org/content/rep  
  -DarchetypeVersion=1.3-SNAPSHOT \  
  -DgroupId=wiki-edits \  
  -DartifactId=wiki-edits \  
  -Dversion=0.1 \  
  -Dpackage=wikiedits \  
  -DinteractiveMode=false
```

构建 JAR 包

```
mvn clean package -Pbuild-jar
```

5. DataStream API

转换操作

- 基本转换
 - `map(T): O`
 - `filter(T): Boolean`
 - `flatMap(T, Collector[O]): Unit`
- 基于 **KeyedStream** 的转换
 - `keyBy: DataStream => KeyedStream`
 - 滚动聚合: `sum, min, max, minBy, maxBy`
 - `reduce(T, T): T`
- 多流转换
 - `union: FIFO`
 - `connect: DataStream => ConnectedStream`
 - `coMap, coFlatMap`
 - 结合 `keyBy` 和 `broadcast`
 - e.g. 当温度>threshold 且 烟雾指数很高时, 发出火灾警报
 - `split(OutputSelector[IN]) => SplitStream`
 - `select(IN): Iterable[String]`
- 分发转换
 - 随机: `shuffle`
 - 轮流: `rebalance`
 - 重调: `rescale` (部分后继任务)
 - 广播: `broadcast`
 - 全局: `global` (下游算子的第一个并行任务)
 - 自定义: `partitionCustom(Partitioner)`

设置并行度

- 覆盖环境默认并行度: 无法通过提交客户端控制应用并行度
- 覆盖算子默认并行度

类型

- 支持的数据类型
 - Java & Scala 原始类型
 - Java & Scala 元组
 - Scala 样例类
 - POJO
 - 数组、列表、映射、枚举以及其他特殊类型

- 为数据类型创建类型信息 `val stringType: TypeInfoInformation[String] = Types.STRING`
- 显式提供类型信息: `ResultTypeQueryable`

定义键值和引用字段

- 字段位置 (Tuple)
- 字段表达式 (Tuple, POJO, case class)
- 键值选择器: `getKey(IN): KEY`

实现函数

- 函数类 (或匿名类)
- Lambda 函数
- 富函数: `RichMapFunction`、`RichFlatMapFunction`
 - `open`: 初始化方法, 常用于一次性设置工作, 如连接外部系统
 - `close`: 终止方法, 常用于清理和释放资源

导入外部和 **Flink** 依赖

- [推荐] 将全部依赖打进应用的 JAR 包
- 将依赖的 JAR 包放在设置 Flink 的 `.lib` 目录中

6. 基于时间和窗口的算子

配置时间特性

- **StreamExecutionEnvironment**
 - **ProcessingTime**: 处理机器的系统时钟，极低延迟
 - **EventTime**: 数据自身包含的信息（时间戳），处理乱序到达
 - **IngestionTime**: 把在数据源算子的处理时间作为事件时间，并自动生成水位线
- 分配时间戳和生成水位线
 - 在数据源完成: **SourceFunction**
 - 周期分配器: **AssignerWithPeriodicWatermarks**
 - 时间间隔: `ExecutionConfig.setAutoWatermarkInterval(default = 200ms)`
 - 如果 `getCurrentWatermark` 返回非空，且大于上一个水位线时间戳，就发出新水位线
 - `assignAscendingTimeStamps` : 输入元素时间戳单调增加
 - `BoundedOutOfOrdernessTimeStampExtractor` : 接受最大预期延迟
 - 定点分配器: **AssignerWithPunctuatedWatermarks**
 - `extractTimestamp` -> `checkAndGetNextWatermark`
- 水位线: 平衡延迟和结果的完整性

处理函数

- `ProcessFunction` , `KeyedProcessFunction` , `CoProcessFunction` , `ProcessJoinFunction` , `BroadcastProcessFunction` , `KeyedBroadcastProcessFunction` , `ProcessWindowFunction` , `ProcessAllWindowFunction`
- `KeyedProcessFunction[KEY, IN, OUT]`
 - 实现 `RichFunction`: `open()` , `close()` , `getRuntimeContext()`
 - `processElement(v: IN, ctx: Context, out: Collector[OUT])`
 - 每个记录都调用一次
 - 可以通过 `Context` 对象访问时间戳、当前记录的键值、`TimerService`
 - `onTimer(timestamp: Long, ctx: Context, out: Collector[OUT])`
 - 回调函数，注册的计时器触发时被调用
- 时间服务和计时器 **TimerService**
 - `currentProcessingTime(): Long`

- `currentWatermark(): Long`
- `registerProcessingTimeTimer(timestamp: Long): Unit` // 处理时间计时器（机器时间）
- `registerEventTimeTimer(timestamp: Long): Unit` // 事件事件计时器（水位线）
- `deleteProcessingTimeTimer(timestamp: Long): Unit`
- `deleteEventTimeTimer(timestamp: Long): Unit`
- 对于每个键值和时间戳，只能注册一个计时器
- e.g. 如果某个传感器的温度在1秒内的处理时间内持续上升，则发出警告
- 向副输出发送数据 **OutputTag[X]**
 - e.g. 在遇到读数温度低于 32°F 的记录，向副输出发送警告
- **CoProcessFunction**
 - e.g. `sensor_2` 转发 10秒，`sensor_7` 转发 1分钟

窗口算子

- 定义窗口算子
 - 键值分区：


```
stream.keyBy(...).window(...).reduce/aggregate/process(...)
```
 - 非键值分区（不支持并行）：


```
stream.windowAll(...).reduce/aggregate/process(...)
```
- 内置窗口算子
 - 滚动窗口（默认对齐，可以指定偏移量）
 - `.window(TumblingEventTimeWindows.of(Time.seconds(1)))`
 - `.window(TumblingProcessingTimeWindows.of(Time.seconds(1)))`
 - 简写： `.timeWindow(time.seconds(1))`
 - 滑动窗口（窗口大小、滑动间隔）
 - `.window(SlidingEventTimeWindows.of(Time.hours(1), Time.minutes(15)))`
 - `.window(SlidingProcessingTimeWindows.of(Time.hours(1), Time.minutes(15)))`
 - 简写： `.timeWindow(Time.hours(1), Time.minutes(15))`
 - 会话窗口（非活动间隔）
 - `.window(EventTimeSessionWindows.withGap(Time.minutes(15)))`
 - `.window(ProcessingTimeSessionWindows.withGap(Time.minutes(15)))`
- 在窗口上应用函数（增量/全量）
 - **ReduceFunction**：简单聚合，e.g. 最低温度

- **AggregateFunction**: `createAccumulator` , `add` , `getResult` , `merge` , e.g. 平均温度
- **ProcessWindowFunction**: 全量元素复杂计算, e.g. 5秒滚动分组, 计算最低温和最高温
 - `process(key: KEY, ctx: Context, vals: Iterable[IN], out: Collector[OUT])`
- 增量聚合与 **ProcessWindowFunction**
 - **Context** 对象可以访问窗口的元数据或状态, 可以连接在 **Reduce/Process** 后面
 - 窗口触发器触发时, 传递给 `process` 的 `Iterable.length == 1`

```
input
  .keyBy(...)
  .timeWindow(...)
  .reduce(
    incrAggregator: ReduceFunction[IN] || AggregateFunction[
      function: ProcessWindowFunction[IN, OUT, K, W]
  )
```

自定义窗口算子

- 窗口的生命周期
 - 窗口创建: **WindowAssigner** 首次向它分配元素时
 - 窗口状态: 窗口内容、窗口对象、触发器计时器、触发器中的自定义状态
 - 窗口删除: 窗口结束时间到达时, 需要清除自定义触发器状态 (状态泄露)
- 窗口分配器
 - **WindowAssigner** 将到来的元素分配给哪些窗口 (0/1/n), e.g. 每30秒滚动窗口
 - **MergingWindowAssigner**: 对已有窗口合并, 例如 **EventTimeSessionWindows** 分配器
- 触发器 (optional)
 - 默认触发器: 处理时间或水位线超过窗口结束边界时间
 - 提前触发: 水位线到达结束时间戳前, 计算, 并发出早期结果 (保守水位线+低延迟结果)
 - **TriggerResult**: `CONTINUE` , `FIRE` , `PURGE` , `FIRE_AND_PURGE`
- 移除器 (optional): `evictBefore`, `evictAfter`
 - `elements: Iterable`, `size: Int`, `window: W`, `context: EvictorContext`

基于时间的双流 Join

- 基于间隔的 Join: 相同键值 && 时间戳不超过指定间隔, e.g.
`between(<lower-b>, <upper-b>)`
- 基于窗口的
Join: `input1.join(input2).where(p1).equalTo(p2).window(...).apply(...)`

.png>)

处理迟到数据

- 丢弃迟到事件: **default**
- 重定向迟到事件: 副输出重定向 -> 后续处理 (如定期回填操作 **backfill process**)
- 基于迟到事件更新结果: 延迟容忍度 `allowedLateness()`, 根据状态标识 **first / update**

7. 有状态算子和应用

实现有状态函数

- 在 **RuntimeContext** 中声明键值分区状态
 - 键值分区状态原语（必须作用在 **KeyedStream**）
 - `ValueState[T]` , `ListState[T]` , `MapState[K, V]` : 支持读取、更新, 支持 `State.clear()`
 - `ReducingState[T]` , `AggregatingState[I, O]` : `add` 后立即返回, 支持 `State.clear()`
 - e.g. 相邻温度变化超过给定阈值时, 发出警报
 - **RichFunction**: `open` 初始化, 注册 `StateDescriptor`, 状态引用对象 作为 普通成员变量
 - 单个 `ValueState` 简写: `mapWithState||flatMapWithState((IN, Option[S]))`
- 通过 **ListCheckpointed** 接口实现算子列表状态
 - **ListCheckpointed**: 列表结构允许对使用了就算子状态的函数修改并行度
 - `snapshotState(checkpointId: Long, timestamp: Long): java.util.List[T]` // 返回状态快照列表
 - `restoreState(java.util.List[T] state): Unit` // 恢复函数状态（启动/故障恢复）
 - e.g. 每个函数并行实例内, 统计该分区数据超过某一阈值的温度值数目
- 使用联结的广播状态
 - e.g. 一条规则流 + 一条需要应用这些规则的事件流
 - 步骤:
 - 调用 `DataStream.broadcast()` 方法创建 `BroadcastStream` 并提供 `MapStateDescriptor` 对象
 - 将 `BroadcastStream` 和一个 `DataStream / KeyedStream` 联结起来 (`connect`)
 - 在联结后的数据流上应用函数:
`(Keyed)BroadcastProcessFunction`
- 使用 **CheckpointedFunction** 接口
 - **CheckpointedFunction**: 用于指定由状态函数的最底层接口
 - `initializeState()`: 创建并行实例时被调用
 - `snapshotState()`: 在生成检查点之前调用
 - e.g. 分别利用键值分区状态和算子状态, 统计每个键值分区和每个算子实例高温读数
- 接收检查点完成通知
 - `CheckpointListener.notifyCheckpointComplete(long checkpointid)`

- 会在 **JobManager** 将检查点注册为已完成时调用，但不保证每个完成的检查点都会调用

为有状态的应用开启故障恢复

- 检查点间隔：较短的间隔会为常规处理带来较大的开销，但由于恢复时要重新处理的数据量较小，所以恢复速度会更快
- 其他配置选项：一致性保障的选择、可同时生成的检查点数量、.....

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.enableCheckpointing(10000L) // 10s
```

确保有状态应用的可维护性

```
// 算子唯一标识
val alerts: DataStream[(String, Double, Double)] = keyedSensorD
    .flatMap(new TemperatureAlertFunction(1.1))
    .uid("TempAlert")

// 最大并行度（具有键值分区状态的算子）
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setMaxParallelism(512) // 应用级别
val alerts: DataStream[(String, Double, Double)] = keyedSensorD
    .flatMap(new TemperatureAlertFunction(1.1))
    .setMaxParallelism(1024) // 算子级别，覆盖应用级别
```

有状态应用的性能及鲁棒性

- 选择状态后端

状态后端	特性	本地	检查点
MemoryStateBackend	仅作开发/调试	JVM 内存，延迟低，但 OOM/GC	JobManager 堆内存
FsStateBackend	本地内存+故障容错	JVM 内存，延迟低，但 OOM/GC	远程持久化文件系统
RocksDBStateBackend	用于非常大的状态	本地 RocksDB 实例，序列化读写	远程持久化文件系统

- 选择状态原语
 - MapState[X, Y] 优于 ValueState[HashMap[X, Y]]
 - ListState[X] 优于 ValueState[List[X]]
 - 建议每次函数调用只更新一次状态（检查点需要和函数调用同步）
- 防止状态泄露
 - 清除无用状态（键值域变化）：注册针对未来某个事件点的计时器（窗口 Trigger、处理函数）

更新有状态应用

生成保存点，停用该应用，重启新版本

- 保持现有状态更新应用
- 从应用中删除状态
- 修改算子的状态
 - (✓) 通过更改状态的数据类型，e.g. ValueState[Int] => ValueState[Double], Avro 类型支持
 - (x) 通过更改状态原语类型，e.g. ValueState[List[String]] => ListState[String], 不支持

查询式状态

- 服务进程：QueryableStateClient, QueryableStateClientProxy, QueryableStateServer
- 对外暴露可查询式状态
 - 通用方法：StateDescriptor.setQueryable(String)
 - 简便方法：数据流.keyBy().setQueryable(String) 添加可查询式状态的数据汇
- 从外部系统查询状态
 - QueryableStateClient(tmHostname, proxyPort).getKvState()

- e.g. Flink 应用状态查询仪表盘

8. 读写外部系统

应用的一致性保障

- 数据源连接器
 - 基于文件的连接器（文件字节流的读取偏移）
 - **Kafka**连接器（消费topic分区的读取偏移）
- 数据汇连接器
 - 幂等性写：故障恢复过程中，临时性不一致（e.g. 将相同的键值对插入HashMap）
 - 事务性写：只有在上次成功的检查点之前计算的结果才被写入（延迟）
 - **WAL 数据汇（write-ahead log）**：将所有结果记录写入应用状态，并在检查点完成后发送
 - **2PC 数据汇（two-phase commit）**：收到检查点完成通知后，提交事务真正写入结果

	不可重置数据源	可重置数据源
任意数据汇	至多一次	至少一次
幂等性数据汇	至多一次	精确一次
WAL 数据汇	至多一次	至少一次
2PC 数据汇	至多一次	精确一次

内置连接器

Kafka 数据源

```
// org.apache.flink.flink-connector-kafka_2.12 (v1.7.1)
val properties = new Properties()
properties.setProperty("bootstrap.servers", "localhost:9092")
properties.setProperty("group.id", "test")

val stream: DataStream[String] = env.addSource(
    new FlinkKafkaConsumer[String](
        "topic", // topic: 单个/列表/正则表达式
        new SimpleStringSchema(), // (Keyed)DeserializationSchema
        properties)) // properties

FlinkKafkaConsumer.assignTimestampAndWatermark() // 时间戳
- AssignerWithPeriodicWatermark
- AssignerWithPunctuatedWatermark

FlinkKafkaConsumer.setStartFromGroupoffsets() // 最后读
FlinkKafkaConsumer.setStartFromEarliest() // 每个分
FlinkKafkaConsumer.setStartFromLatest() // 每个分
FlinkKafkaConsumer.setStartFromTimestamp(long) // 时间戳
FlinkKafkaConsumer.setStartFromSpecificOffsets(Map) // 为所有

// 自动发现满足正则表达式的新主题
// properties.setProperty("flink.partition-discovery.interval-m")
```

Kafka 数据汇

```
// org.apache.flink.flink-connector-kafka_2.12 (v1.7.1)
val stream: DataStream[String] = ...
val myProducer = new FlinkKafkaProducer[String](
    "localhost:9092", // Kafka Broker 列表 (,)
    "topic", // 目标 topic
    new SimpleStringSchema) // 序列化的 Schema
stream.addSink(myProducer)
```

- 至少一次保障
 - Flink检查点开启，且所有数据源可重置
 - 如果数据汇连接器写入不成功，则会抛出异常（default）
 - 数据汇连接器要在检查点完成前等待Kafka确认写入完毕（default）
- 精确一次保障（v0.11）
 - Semantic.NONE
 - Semantic.AT_LEAST_ONCE (default)
 - Semantic.EXACTLY_ONCE: 全部消息追加到分区日志，并在事务提交后标记为已提交（延迟）

- 自定义分区和写入消息时间戳
 - 自定义分区：提供 `KeyedSerializationSchema`，`FlinkKafkaPartitioner=null` 禁用默认分区器
 - 写入时间戳（v0.10）：`setWriteTimestamp ToKafka(true)`

文件系统数据源

```
val lineReader = new TextInputFormat(null)
val lineStream: DataStream[String] = env.readFile[String](
  lineReader,                                     // FileInputFor
  "hdfs:///path/to/my/data",                     // 读取路径
  FileProcessing.Mode.PROCESS_CONTINUOUSLY,       // 处理模式，或
  30000L)                                         // 监控间隔（ms）
```

文件系统数据汇

```
val input: DataStream[String] = ...
val sink: StreamingFileSink[String] = StreamingFileSink
  .forRowFormat(                                  // 行编码
    new Path("/base/path"),
    new SimpleStringEncoder[String]("UTF-8"))
  .build()
input.addSink(sink)
```

- 桶的选择 `BucketAssigner` (default = `DateTimeBucketAssigner`)
 - 路径规则：`[base-path]/[bucket-part]/part-[task-idx]-[id]`
 - `RollingPolicy`：何时创建一个新的分块文件（默认文件大小超过128M或事件超过60秒）
- 两种分块文件写入模式
 - 行编码（row encoding）：将每条记录单独写入分块文件中（`SimpleStringEncoder`）
 - 批量编码（bulk encoding）：记录会被攒成批，然后一次性写入，如果 Apache Parquet

```
val input: DataStream[String] = ...
val sink: StreamingFileSink[String] = StreamingFileSink
  .forBulkFormat(                                 // 批量编码
    new Path("/base/path"),
    ParquetAvroWriter.forSpecificRecord(classOf[AvroPojo]))
  .build()
input.addSink(sink)
```

Cassandra 数据汇

- 针对元组

```
// org.apache.flink.flink-connector-cassandra_2.12 (v1.7.1)
// 定义 Cassandra 示例表
CREATE KEYSPACE IF NOT EXISTS example
    WITH replication = {'class': 'SimpleStrategy', 'replication_
CREATE TABLE IF NOT EXISTS example.sensors (
    sensorId VARCHAR,
    temperature FLOAT,
    PRIMARY KEY(sensorId)
);

// 针对元组的 Cassandra 数据汇
val readings: DataStream[(String, Float)] = ...
val sinkBuilder: CassandraSinkBuilder[(String, Float)] =
    CassandraSink.addSink(readings)
sinkBuilder.setHost("localhost").setQuery(
    "INSERT INTO example.sensor(sensorId, temperature) VALUES (?
    .build()
```

- 针对 POJO 类型

```
// 针对 POJO 类型创建 Cassandra 数据汇
val readings: DataStream[(String, Float)] = ...
CassandraSink.addSink(readings).setHost("localhost").build()

// 添加了 Cassandra Object Mapper 注释的 POJO 类
@Table(keyspace = "example", name = "sensors")
class SensorReading(
    @Column(name = "sensorId") var id: String,
    @Column(name = "temperature") var temp: Float) {

    def this() = this("", 0.0)
    def setId(id: String): Unit = this.id = id
    def getId: String = id
    def setTemp(temp: Float): Unit = this.temp = temp
    def getTemp: Float = temp
}
```

- setClusterBuilder(ClusterBuilder): 管理和 Cassandra 的连接
- setHost(String, [Int])
- setQuery(String)
- setMapperOptions(MapperOptions)

- `enableWriteAheadLog([CheckpointCommitter])`: 开启 WAL，提供精确一次输出保障

实现自定义数据源函数

- `SourceFunction` 和 `RichSourceFunction`: 定义非并行的数据源连接器（单任务运行）
- `ParallelSourceFunction` 和 `ParallelRichSourceFunction`: 定义并行的数据源连接起
- `(Parallel)SourceFunction`: `run()` 执行记录读入，`cancel()` 在取消或关闭时调用
- 可重置的数据源函数
 - 实现 `CheckpointedFunction` 接口，并把读取偏移和相关元数据信息存入状态
 - `run()` 不会再检查点生成过程中推进偏移/发出数据：
`SourceContext.getCheckpointLock()`
- 数据源函数、时间戳及水位线
 - `SourceContext`
 - `collectWithTimestamp(record: T, timestamp: Long): Unit`
 - `def emitWatermark(watermark: Watermark): Unit`
 - 标记为暂时空闲：`SourceContext.markAsTemporarilyIdle()`
- **`SourceFunction`**

```
// 从 0 数到 Long.MaxValue 的 SourceFunction
class CountSource extends SourceFunction[Long] {
  var isRunning: Boolean = true

  override def run(ctx: SourceFunction.SourceContext[Long]): Unit = {
    var cnt: Long = -1
    while (isRunning && cnt < Long.MaxValue) {
      // increment cnt
      cnt += 1
      ctx.collect(cnt)
    }
  }

  override def cancel(): Unit = isRunning = false
}
```

- 可重置的 **`SourceFunction`**

```

// 可重置的 SourceFunction
class ReplayableCountSource
    extends SourceFunction[Long] with CheckpointedFunction {

    var isRunning: Boolean = true
    var cnt: Long = _
    var offsetState: ListState[Long] = _

    override def run(ctx: SourceFunction.SourceContext[Long]): Unit = {
        while (isRunning && cnt < Long.MaxValue) {
            ctx.getCheckpointLock.synchronized {
                // increment cnt
                cnt += 1
                ctx.collect(cnt)
            }
        }
    }

    override def cancel(): Unit = isRunning = false

    override def snapshotState(snapshotCtx: FunctionSnapshotContext): ListState[Long] = {
        // remove previous cnt
        offsetState.clear()
        // add current cnt
        offsetState.add(cnt)
    }

    override def initializeState(initCtx: FunctionInitializationContext): ListState[Long] = {
        // obtain operator list state to store the current cnt
        val desc = new ListStateDescriptor[Long]("offset", classOf[Long])
        offsetState = initCtx.getOperatorStateStore.getListState(desc)
        // initialize cnt variable from the checkpoint
        val it = offsetState.get()
        cnt = if (null == it || !it.iterator().hasNext) {
            -1L
        } else {
            it.iterator().next()
        }
    }
}

```

实现自定义数据汇函数

- SinkFunction.invoke(value: IN, ctx: Context): e.g. 套接字
- 幂等性数据汇连接器: e.g. Derby

- 事务性数据汇连接器：实现 CheckpointListener 接口
 - GennericWriteAheadSink：至少一次
 - 参数：CheckopintCommitter, TypeSerializer, 任务id
 - 方法：boolean sendValues(Iterable\ values, long chkpntId, long timestamp)
 - TwoPhaseCommitSinkFunction：精确一次
 - 要求
 - 外部数据汇支持事务、检查点间隔期事务开启、事务接到完成通知后提交
 - 数据汇需要在进程故障时进行事务恢复、提交事务的操作是幂等的
 - 方法：
 - beginTransaction()
 - invoke(txn: TXN, value: IN, context: Context[...])
 - preCommit(txn: TXN)、commit(txn: TXN)
 - abort(txn: TXN)
- SinkFunction


```
// nc -l localhost 9091
// write the sensor readings to a socket
val reading: DataStream[SensorReading] = ...
readings.addSink(new SimpleSocketSink("localhost", 9191).setPara

// Writes a stream of [[SensorReading]] to a socket.
class SimpleSocketSink(val host: String, val port: Int)
    extends RichSinkFunction[SensorReading] {

    var socket: Socket = _
    var writer: PrintStream = _

    override def open(config: Configuration): Unit = {
        // open socket and writer
        socket = new Socket(InetAddress.getByName(host), port)
        writer = new PrintStream(socket.getOutputStream)
    }

    override def invoke(
        value: SensorReading,
        ctx: SinkFunction.Context[_]): Unit = {
        // write sensor reading to socket
        writer.println(value.toString)
        writer.flush()
    }

    override def close(): Unit = {
        // close writer and socket
        writer.close()
        socket.close()
    }
}
```

- **IdempotentSinkFunction**

```

// write the converted sensor readings to Derby.
val reading: DataStream[SensorReading] = ...
readings.addSink(new DerbyUpsertSink)

// Sink that upserts SensorReadings into a Derby table
class DerbyUpsertSink extends RichSinkFunction[SensorReading] {

    var conn: Connection = _
    var insertStmt: PreparedStatement = _
    var updateStmt: PreparedStatement = _

    override def open(parameters: Configuration): Unit = {
        // connect to embedded in-memory Derby
        val props = new Properties()
        conn = DriverManager.getConnection("jdbc:derby:memory:flinkE
        // prepare insert and update statements
        insertStmt = conn.prepareStatement(
            "INSERT INTO Temperatures (sensor, temp) VALUES (?, ?)")
        updateStmt = conn.prepareStatement(
            "UPDATE Temperatures SET temp = ? WHERE sensor = ?")
    }

    override def invoke(r: SensorReading, context: Context[_]): Un
        // set parameters for update statement and execute it
        updateStmt.setDouble(1, r.temperature)
        updateStmt.setString(2, r.id)
        updateStmt.execute()
        // execute insert statement if update statement did not upda
        if (updateStmt.getUpdateCount == 0) {
            // set parameters for insert statement
            insertStmt.setString(1, r.id)
            insertStmt.setDouble(2, r.temperature)
            // execute insert statement
            insertStmt.execute()
        }
    }

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

```

- WAL

```

// print to standard out with a write-ahead log.
// results are printed when a checkpoint is completed.
val readings: DataStream[SensorReading] = ...
readings.transform("WriteAheadSink", new StdOutWriteAheadSink).s

// Write-ahead sink that prints to standard out and
// commits checkpoints to the local file system.
class StdOutWriteAheadSink extends GenericWriteAheadSink[(String, Double)] {
  // CheckpointCommitter that commits checkpoints to the local
  // file system
  new FileCheckpointCommitter(System.getProperty("java.io.tmpdir"),
    // Serializer for records
    createTypeInfo[(String, Double)].createSerializer(new
    // Random JobID used by the CheckpointCommitter
    UUID.randomUUID().toString) {

    override def sendValues(
      readings: Iterable[(String, Double)],
      checkpointId: Long,
      timestamp: Long): Boolean = {

      for (r <- readings.asScala) {
        // write record to standard out
        println(r)
      }
      true
    }
  }
}

```

- 2PC

```

class TransactionalFileSink(val targetPath: String, val tempPath
    extends TwoPhaseCommitSinkFunction[(String, Double), String,
        createTypeInformation[String].createSerializer(new Execution
        createTypeInformation[Void].createSerializer(new Execution

var transactionWriter: BufferedWriter = _

/** Creates a temporary file for a transaction into which the
    * written.
    */
override def beginTransaction(): String = {

    // path of transaction file is constructed from current time
    val timeNow = LocalDateTime.now(ZoneId.of("UTC"))
        .format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
    val taskIdx = this.getRuntimeContext.getIndexOfThisSubtask
    val transactionFile = s"$timeNow-$taskIdx"

    // create transaction file and writer
    val tFilePath = Paths.get(s"$tempPath/$transactionFile")
    Files.createFile(tFilePath)
    this.transactionWriter = Files.newBufferedWriter(tFilePath)
    println(s"Creating Transaction File: $tFilePath")

    // name of transaction file is returned to later identify th
    transactionFile
}

/** Write record into the current transaction file. */
override def invoke(transaction: String, value: (String, Double),
    context: Context[_]): Unit = {

    transactionWriter.write(value.toString)
    transactionWriter.write('\n')
}

/** Flush and close the current transaction file. */
override def preCommit(transaction: String): Unit = {
    transactionWriter.flush()
    transactionWriter.close()
}

/** Commit a transaction by moving the pre-committed transacti
    * to the target directory.
    */
override def commit(transaction: String): Unit = {
    val tFilePath = Paths.get(s"$tempPath/$transaction")
    // check if the file exists to ensure that the commit is ide

```

```

    if (Files.exists(tFilePath)) {
        val cFilePath = Paths.get(s"$targetPath/$transaction")
        Files.move(tFilePath, cFilePath)
    }
}

/** Aborts a transaction by deleting the transaction file. */
override def abort(transaction: String): Unit = {
    val tFilePath = Paths.get(s"$tempPath/$transaction")
    if (Files.exists(tFilePath)) {
        Files.delete(tFilePath)
    }
}
}

```

异步访问外部系统

- AsyncFunction
 - orderedWait(): 按照数据记录顺序发出结果的异步算子
 - unorderedWait(): 只能让水位线和检查点分隔符保持对齐
- 示例中为每条记录创建 JDBC 连接，实际应用中需要避免

• orderedWait

```

// look up the location of a sensor from a Derby table with asyn
val readings: DataStream[SensorReading] = ...
val sensorLocations: DataStream[(String, String)] = AsyncDataStr
    .orderedWait(
        readings,
        new DerbyAsyncFunction,
        5, TimeUnit.SECONDS,           // timeout requests after 5 seco
        100)                          // at most 100 concurrent reques

```

• DerbyAsyncFunction

```

// AsyncFunction that queries a Derby table via JDBC in a non-bl
class DerbyAsyncFunction extends AsyncFunction[SensorReading, (S

    // caching execution context used to handle the query threads
    private lazy val cachingPoolExecCtx =
        ExecutionContext.fromExecutor(Executors.newCachedThreadPool(
    // direct execution context to forward result future to callba
    private lazy val directExecCtx =
        ExecutionContext.fromExecutor(
            org.apache.flink.runtime.concurrent.Executors.directExecut

/** Executes JDBC query in a thread and handles the resulting
 * with an asynchronous callback. */
override def asyncInvoke(
    reading: SensorReading,
    resultFuture: ResultFuture[(String, String)]): Unit = {

    val sensor = reading.id

    // get room from Derby table as Future
    val room: Future[String] = Future {
        // Creating a new connection and statement for each record
        // Note: This is NOT best practice!
        // Connections and prepared statements should be cached.
        val conn = DriverManager
            .getConnection("jdbc:derby:memory:flinkExample", new Pro
        val query = conn.createStatement()

        // submit query and wait for result. this is a synchronous
        val result = query.executeQuery(
            s"SELECT room FROM SensorLocations WHERE sensor = '$sens

        // get room if there is one
        val room = if (result.next()) {
            result.getString(1)
        } else {
            "UNKNOWN ROOM"
        }

        // close resultset, statement, and connection
        result.close()
        query.close()
        conn.close()

        // sleep to simulate (very) slow requests
        Thread.sleep(2000L)

        // return room

```

```
    room
  }(cachingPoolExecCtx)


  // apply result handling callback on the room future
  room.onComplete {
    case Success(r) => resultFuture.complete(Seq((sensor, r)))
    case Failure(e) => resultFuture.completeExceptionally(e)
  }(directExecCtx)
}
}
```

9. 搭建 Flink 运行流式应用

部署模式

独立集群

- 独立集群至少包含：一个主进程、一个 TaskManager 进程
 - 主进程（线程）：Dispatcher、ResourceManger
 - TaskManager：将自己注册到 ResourceManager 中
- 主进程和工作进程不会因为故障而重启
 - 只要有足够多的剩余 slot，作业就可以从工作进程故障中恢复
 - 从主进程故障中恢复作业需要 HA 设置

 向 Flink 独立集群提交应用.png>)

```
tar xvfz flink-1.11.0-bin-scala_2.11.tgz
cd flink-1.11.0
./bin/start-cluster.sh # http://localhost:8081
./bin/stop-cluster.sh
```

Docker

```
# 启动主进程
docker run -d --name flink-jobmanager \
-e JOB_MANAGER_RPC_ADDRESS=jobmanager \
-p 8081:8081 flink:1.7 jobmanager


# 启动工作进程
docker run -d --name flink-taskmanager-1 \
--link flink-jobmanager:jobmanager \
-e JOB_MANAGER_RPC_ADDRESS=jobmanager flink:1.7 taskmanager
```

Apache Hadoop YARN

- 作业模式（job mode）：运行单个任务，作业结束后集群停止
- 会话模式（session mode）：启动一个长时间运行的 Flink 集群，可以运行作业，需要手动停止


```
# job mode
./bin/flink run -m yarn-cluster ./path/to/job.jar

# session mode
./bin/yarn-session.sh # start a Flink YARN session
./bin/flink run ./path/to/job.jar # submit job to session
```

 作业模式（job mode）启动.png>)

Kubernetes

- **Deployment:** 主进程 Pod、工作进程 Pod
- **Service:** 将主进程 Pod 的端口暴露给工作进程所在的 Pod

```
# 注册在 Kubernetes
kubectl create -f master-deployment.yaml
kubectl create -f worker-deployment.yaml
kubectl create -f master-service.yaml

kubectl get deployments # 查看所有 Deployment 的状态
kubectl port-forward deployment/flink-master 8081:8081 # 端口转发

# 关闭 Flink 集群
kubectl delete -f master-deployment.yaml
kubectl delete -f worker-deployment.yaml
kubectl delete -f master-service.yaml
```

高可用性设置

依赖于 Apache ZooKeeper 以及某种持久化远程存储（HDFS、NFS、S3）

```
# ./conf/flink-conf.yaml
# 通过 ZooKeeper 开启 HA 模式
high-availability: zookeeper

# 提供 ZooKeeper Quorum 的服务器列表
high-availability.zookeeper.quorum: address1:2181[,...],addressX

# 设置作业员数据的远程存储位置
high-availability.storageDir: hdfs:///flink/recovery

# (optional) 在 ZooKeeper 中为全部 Flink 集群设置基础路径
# 将 Flink 和其他使用 ZooKeeper 集群的框架进行隔离
high-availability.zookeeper.path.root: /flink
```

独立集群

- 无须依赖资源提供者，需要后备 Dispatcher 和 TaskManager 进程
- ZooKeeper 会在所有 Dispatcher 中选出一个负责执行应用的领导者

```
# ./conf/flink-conf.yaml
# (optional) 在 Zookeeper 中为 Flink 集群设置路径
# 将多个 Flink 集群互相隔离，集群 ID 是查找故障集群元数据的必要信息
high-availability.cluster-id: /cluster-1
```

YARN

```
# yarn-site.xml
# ApplicationMaster 尝试执行的最大次数，default=2（即重启一次）
yarn.resourcemanager.am.max-attempts: 4
```

```
# ./conf/flink-conf.yaml
# 应用最多重启 3 次（包括首次启动）
# 该值必须小于或等于配置的最大尝试次数
yarn.application-attempts: 4
```

Kubernetes

- 主进程HA: `./conf/flink-conf.yaml`
 - ZooKeeper Quorum 节点的主机名，持久化存储路径，Flink 集群ID

集成 Hadoop 组件

- 为 Flink 添加 Hadoop 依赖的方式
 - 使用针对特定 Hadoop 版本构建的 Flink 二进制发行版
 - 使用特定版本的 Hadoop 构建 Flink (mvn)
 - 使用不带 Hadoop 的 Flink 发行版，并手动配置 Classpath

文件系统配置

FS	URI协议	备注
本地文件系统	file://	包括本地挂载的网络文件系统 (NFS、SAN)
Hadoop HDFS	hdfs://	
Amazon S3	s3://	分别基于 Apache Hadoop 和 Presto 实现
OpenStack Swift FS	swift://	swift-connector JAR

系统配置

- **Java** 和类加载：默认使用 PATH，可以通过配置文件中的 JAVA_HOME 或 env.java.home 指定
- **CPU**：taskmanager.numberOfTaskSlots (默认：每个 TaskManager 一个 slot)
- 内存和网络缓冲
 - jobmanager.heap.size (默认 1GB)
 - taskmanager.heap.size (JVM、网络 Netty、RocksDB 状态后端)
 - 网络：taskmanager.network.memory.fraction (segment-size/min/max)
 - RocksDB：每个列簇需要大约 200MB - 240MB 的堆外内存
- 磁盘存储
 - io.tmp.dirs (默认为 java.io.tmpdir 或 linux 下的 /tmp 目录)
 - blob.storage.directory (常用于大文件，例如应用 jar 包)
 - state.backend.rocksdb.localdir (默认为 io.tmp.dirs)
- 检查点和状态后端
 - state.backend: async/incremental
 - state.checkpoints.dir, state.savepoints.dir
 - 配置本地恢复：state.backend.local-recovery=true (本地状态副本)

- 安全性
 - keytabs（首选）
 - Hadoop委托令牌：会在一段时间后过期

10. Flink 和流式应用运维

运行并管理流式应用

保存点

```
/savepointes/savepoint-:shortjobid-:savepointid/_metadata # 元数据
/savepointes/savepoint-:shortjobid-:savepointid/:xxx # 存储
```

通过命令行客户端管理应用

```
# 启动应用
./bin/flink run ~/myApp.jar [args] # 启动应用
./bin/flink run -d ~/myApp.jar # 分离模式
./bin/flink run -p 16 ~/myApp.jar # 指定默认并行度
./bin/flink run -c my.app.MainClass ~/myApp.jar # 指定入口类
./bin/flink run -m myMasterHost:9876 ~/myApp.jar # 提交到特定的集群

# 列出正在运行的应用
./bin/flink list -r

# 生成和清除保存点
./bin/flink savepoint <jobId> [savepointPath]
./bin/flink savepoint -d <savepointPath>

# 取消应用
./bin/flink cancel <jobId>
./bin/flink cancel -s [savepointPath] <jobId> # 取消前生成保存点

# 从保存点启动应用
# 如删除, -n 禁用安全检查
./bin/flink run -s <savepointPath> [options] <jobJar> [args]

# 应用的扩缩容
# 生成保存点, 取消, 以新的并行度重启
./bin/flink modify <jobId> -p <newParallelism>
```

通过 REST API 管理应用

```
# curl REST
curl -X <HTTP-Method> [-d <params>] http://hostname:port/v1/<RES
curl -X GET http://localhost:8081/v1/overview

# 管理和监控 Flink 集群
GET /overview # 集群基本信息
GET /jobmanager/config # JobManager 配置
GET /taskmanagers # 列出所有相连的 TaskManager
GET /jobmanager/metrics # 列出 JobManager 可用指标
GET /taskmanagers/<tmId>/metrics # 列出 TaskManager 收集指标
DELETE /cluster # 关闭集群

# 管理和监控 Flink 应用
POST /jars/upload/ # 上传 JAR包
GET /jars # 列出所有已上传 JAR包
DELETE /jars/<jarId> # 删除 JAR包
POST /jars/<jarId>/run # 启动应用
GET /jobs # 列出应用
GET /jobs/<jobId> # 展示应用详细信息
PATCH /jobs/<jobId> # 取消应用
POST /jobs/<jobId>/saveopints # 生成保存点
POST /savepoint-disposal # 删除保存点
PATCH /jobs/<jobId>/rescaling # 应用扩缩容
```

在容器中打包并部署应用

- 构建**Docker**镜像

```
# 针对特定作业构建 Flink Docker 镜像
./flink-container/docker/build.sh \
  -- from-archive <path-to-Flink-1.7.1-archive> \
  -- job-jar <path-to-example-apps-JAR-file> \
  -- image-name flink-book-apps

# 部署在 Docker 的 1个主容器+3个工作容器 上
FLINK_DOCKER_IMAGE_NAME=flink-book-jobs \
  FLINK_JOB=io.github.streamingwithflink.chapter1.AverageSensorR
  DEFAULT_PARALLELISM=3 \
  docker-compose up -d
```

- 在**Kurbernetes**上运行镜像

```
# ./flink-container/kubernetes
kubectl create -f job-cluster-service.yaml
kubectl create -f job-cluster-job.yaml
kubectl create -f task-manager-deployment.yaml
```

控制任务调度

- 控制任务链接
 - 完全禁用：


```
StreamExecutionEnvironment.disableOperatorChaining()
```
 - 禁用算子：


```
input.filter(...).map(new Map1()).map(new Map2()).disableChaining().filter(...)
```
 - 开启新的链接：


```
input.map(new Map1()).map(new Map2()).startNewChain().filter(...)
```

 - `disableChaining` 不能与前后链接，`startNewChain` 可以与后续链接
- 定义 **slot** 共享组
 - `slotSharingGroup(String)`
 - 如果算子所有输入都属于统一共享组则继承，否则加入 **default** 组

调整检查点及恢复

配置检查点

```
val env = StreamExecutionEnvironment = ...
env.enableCheckpointing(10000) // 10sec

// 配置检查点
val cpConfig: CheckpointConfig = env.getCheckpointConfig
cpConfig.setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE) /
cpConfig.setMinPauseBetweenCheckpoints(30000) // 至少不受干扰处理
cpConfig.setMaxConcurrentCheckpoints(3) // 允许同时生成三个检查点
cpConfig.setCheckpointTimeout(30000) // 检查点生成必须在 5min 内完
cpConfig.setFailOnCheckpointingErrors(false) // 不因检查点生成错误

// 支持检查点压缩
env.getConfig.setUseSnapshotCompression(true) // RocksDB 增量检查

// 应用停止后保留检查点
cpConfig.enableExternalizedCheckpoints(
  ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION)
```

配置状态后端

```
val env = StreamExecutionEnvironment = ...
val stateBackend: StateBackend = ...
env.setStateBackend(stateBackend)

// 状态后端
val memBackend = new MemoryStateBackend() // 默认异步, 限制状态 5M
val fsBackend = new FsStateBackend("file:///tmp/ckp", true) // 默
val rocksBackend = new RocksDBStateBackend("file:///tmp/ckp", tr

// 配置选项
val backend: ConfigurableStateBackend = ...
val sbConfig = new Configuration()
sbConfig.setBoolean("state.backend.async", true)
sbConfig.setString("state.savepoints.dir", "file:///tmp/svp")
val configuredBackend = backend.configure(sbConfig)

// RocksDB 参数调整
val backend: RocksDBStateBackend = ...
backend.setPredefinedOptions(PredefinedOptions.SPINNING_DISK_OPT
```

配置故障恢复

- 重启策略
 - fixed-delay: 固定间隔尝试重启某个固定次数
 - failure-rate: 允许在未超过故障率的前提下不断重启 (e.g. 过去十分钟内不超过三次)
 - no-restart: 不会重启, 立即失败

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setRestartStrategy(
  RestartStrategies.fixedDelayRestart(
    5, // 重启尝试次数, 默认 Int.MAX_V
    Time.of(30, TimeUnit.SECONDS) // 尝试之间的延迟, 默认 10sec
  ))
```

- 本地恢复
 - 提高恢复速度, 远程系统作为参照的真实数据 (the source of truth)
 - flink-conf.yaml
 - state.backend.local-recovery: 默认禁用
 - taskmanager.state.local.root-dirs: 一个/多个本地路径

监控 Flink 集群和应用

- **Flink Web UI:** `http://\:8081`
- 指标系统
 - 注册和使用指标：
`MetricGroup (RuntimeContext.getMetrics())`
 - 指标组: `Counter` (计数)、`Gauge` (计算值)、`Histogram` (直方图)、`Meter` (速率)
 - 域和格式指标: 系统域.[用户域].指定名称
(`localhost.taskmanager.512.MyMetrics.myCounter`)
 - `counter =`
`getRuntimeContext.getMetricGroup.addGroup("MyMetrics"`
`).counter("myCounter")`
 - 发布指标: 汇报器 (`reporter`)

```
class PositiveFilter extends RichFilterFunction[Int] {  
  
    @transient private var counter: Counter = _  
    override def open(parameter: Configuration): Unit = {  
        counter = getRuntimeContext.getMetricGroup.counter("droppedE  
    }  
  
    override def filter(value: Int): Boolean = {  
        if (value > 0) {  
            true  
        } else {  
            counter.inc()  
            false  
        }  
    }  
}
```

- 延迟监控

```
env.getConfig.setLatencyTrackingInterval(500L) // ms
```

配置日志行为

- `conf/log4j.properties`
 - `log4j.rootLogger=WARN`
- 其他配置
 - JVM: `-Dlog4j.configuration=parameter`
 - 命令行: `log4j-cli.properties`

- 会话: log4j-yarn-session.properties

```
import org.apache.flink.api.common.functions.MapFunctions
import org.slf4j.LoggerFactory
import org.slf4j.Logger

class MyMapFunction extends MapFunctions[Int, String] {

  Logger LOG = LoggerFactory.getLogger(MyMapFunctions.class)

  override def map(value: Int): String = {
    LOG.info("Converting value {} to string.", value)
    value.toString
  }
}
```

11. 还有什么？

Flink 生态的其他组成部分

- 用于批处理的 DataSet API
- 用于关系型分析的 Table API 及 SQL
- 用于复杂事件处理和模式匹配的 FlinkCEP
 - 金融应用，欺诈检测，监控报警，网络入侵，恶意行为
- 用于图计算的 Gelly