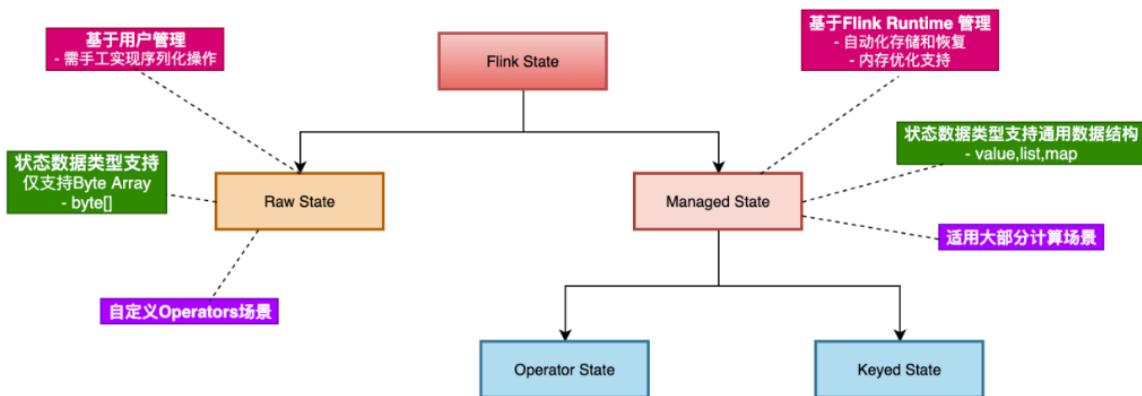


# Flink 1.15.2



# Apache Flink

## 1. Flink 算子状态

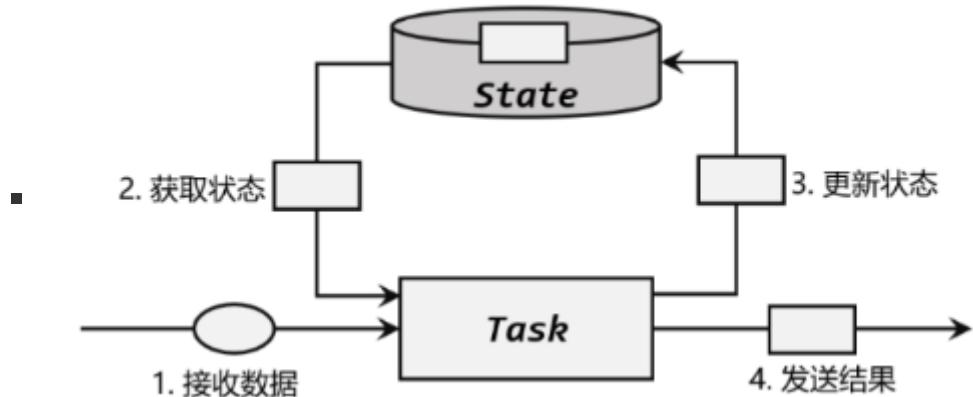


### 1.1. 状态定义

- 在Flink中，算子任务可以分为无状态和有状态两种情况。
  - 无状态算子
    - 无状态的算子任务只需要观察每个独立事件，根据当前输入的数据直接转换输出结果。
    - 如map、filter、flatMap，计算时不依赖其他数据，就都属于无状态的算子。



- 有状态算子
  - 当前数据之外，还需要一些其他数据来得到计算结果。这里的“其他数据”，就是所谓的状态 (state)
  - 比如，做求和 (sum) 计算时，需要保存之前所有数据的和，这就是状态；



- 为什么流式计算需要状态
  - 离线任务失败:
    - 重启任务，然后重新读一遍输入数据，最后把昨天数据重新计算一遍即可。
  - 实时任务失败:
    - 重启任务，然后重新读一遍输入数据，最后把昨天数据重新计算一遍就不可以了。
    - 因为实时任务第一重要的就是时效性，很明显重新计算违背了时效性原则

## 1.2. 托管方式

分类 区别	Managed State	Raw State
状态管理方式	Flink Runtime托管，自动存储、自动恢复，内存管理上有优化	用户自己管理，需要自己进行序列化
数据结构	Flink提供的常用数据结构，如ListState、MapState、ValueState等	字节数组：byte[]
使用场景	大多数情况下均可使用	自定义 Operator 时

### 1.2.1. 原始状态(Raw State)

- 原始状态则是自定义的，相当于就是开辟了一块内存，需要我们自己管理，实现状态的序列化和故障恢复。
- Flink不会对状态进行任何自动操作，也不知道状态的具体数据类型，只会把它当作最原始的字节(Byte)数组来存储。
- 程序员需要花费大量的精力来处理状态的管理和维护。所以只有在遇到托管状态无法实现的特殊需求时，我们才会考虑使用原始状态；
- 一般情况下不推荐使用

### 1.2.2. 托管状态(Managed State)

- 托管状态就是由Flink统一管理的，状态的存储访问、故障恢复和重组等一系列问题都由Flink实现，我们只要调接口就可以；

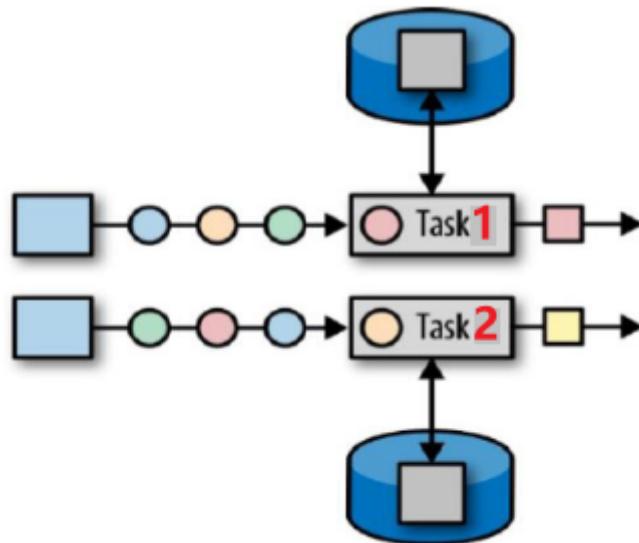
- 托管状态是由Flink的运行时（Runtime）来托管的；在配置容错机制后，状态会自动持久化保存，并在发生故障时自动恢复。
- 当应用发生横向扩展时，状态也会自动地重组分配到所有的子任务实例上。
- Flink提供了值状态（ValueState）、列表状态（ListState）、映射状态（MapState）、聚合状态（AggregateState）等多种结构，内部支持各种数据类型。

## 1.3. 状态类型

- 在Flink中，一个算子任务会按照并行度分为多个并行子任务执行，而不同的子任务会占据不同的任务槽（task slot）。
- 由于不同的slot在计算资源上是物理隔离的，所以Flink能管理的状态在并行任务间是无法共享的，每个状态只能针对当前子任务的实例有效。
- 而很多有状态的操作（比如聚合、窗口）都是要先做keyBy进行按键分区的。
- 按键分区之后，任务所进行的所有计算都应该只针对当前key有效，所以状态也应该按照key彼此隔离。在这种情况下，状态的访问方式又会有所不同。
- 基于这样的想法，我们又可以将托管状态分为两类：算子状态和按键分区状态。

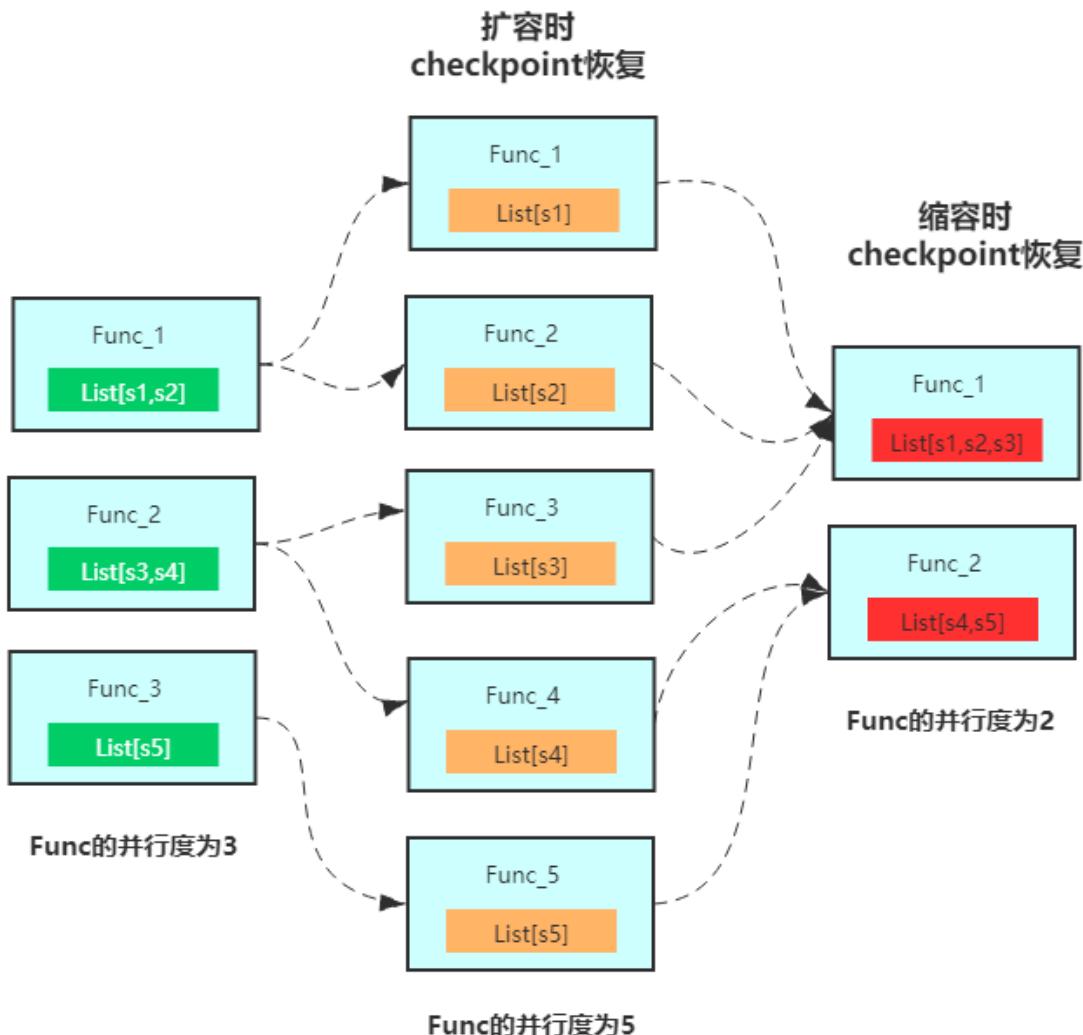
分类区别	Keyed State	Operator State
使用范围	只能用于KeyedStream算子中使用，每个Key对应一个state	可以用于所有算子，常用与Source，比如FlinkKafkaConsumer，一个Operator实例对应一个state
扩缩容模式	Flink把所有键值分为不同的 Key Group，Key Group 是 Flink 重新分配 Keyed State 的最小单元。并发改变时，Flink会以Key Group为单位将键值分配给不同的任务。	当并发改变时，有多种方式来进行重分配，比如ListState使用均匀分配模式，BroadcastState会把状态拷贝到全部新任务上。
访问方式	实现Rich Function，通过getRuntimeContext()返回的RuntimeContext进行获取	实现CheckpointedFunction或者ListCheckpoint的接口
数据结构	ValueState、ListState、ReducingState、AggregatingState、MapState	ListState、BroadcastState等

### 1.3.1. 算子状态(Operator State)



- 状态作用范围限定为当前的算子任务实例，也就是只对当前并行子任务实例有效。

- 这意味着对于一个并行子任务，占据了一个“分区”，它所处理的所有数据都会访问到相同的状态，状态对于同一任务而言是共享的，
- 算子状态可以在所有算子上，使用的时候其实就跟一个本地变量没什么区别——因为本地变量的作用域也是当前任务实例。在使用时，我们还需进一步实现CheckpointedFunction接口。
- 



```

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.state.ListState;
import org.apache.flink.api.common.state.ListStateDescriptor;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.io.File;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello01StateOperator {

```

```
public static void main(String[] args) throws Exception {
    //运行环境
    StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
    environment.setParallelism(2);
    environment.enableCheckpointing(5000);
    environment.getCheckpointConfig().setCheckpointStorage("file://" +
System.getProperty("user.dir") + File.separator + "ckpt");
    //获取数据源
    DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
    //转换并输出
    // source.map(word -> word.toUpperCase()).print();
    //转换需要添加当前subTask处理这个单词的序号并输出
    source.map(new YjxtoOperatorStateFunction()).print();
    //运行环境
    environment.execute();
}
}

class YjxtoOperatorStateFunction implements MapFunction<String, String>,
CheckpointedFunction {

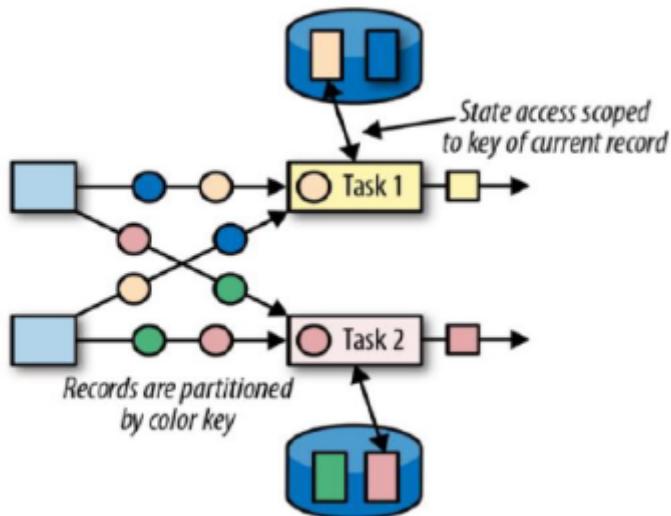
    //声明一个变量记数
    private int count;
    //创建一个状态对象
    private ListState<Integer> countListState;

    @Override
    public String map(String value) throws Exception {
        //更新计数器
        count++;
        return "[" + value.toUpperCase() + "][" + count + "]";
    }

    @Override
    public void snapshotState(FunctionSnapshotContext
functionSnapshotContext) throws Exception {
        //清除一下历史数据
        countListState.clear();
        //保存数据
        countListState.add(count);
        // System.out.println("YjxtoOperatorStateFunction.snapshotState[" +
countListState + "][" + System.currentTimeMillis() + "]");
    }

    @Override
    public void initializeState(FunctionInitializationContext context)
throws Exception {
        //创建对象的描述器
        ListStateDescriptor<Integer> descriptor = new
ListStateDescriptor<Integer>("CountListState", Types.INT);
        //创建对象
        this.countListState =
context.getOperatorStateStore().getListState(descriptor);
    }
}
```

### 1.3.2. 键分区状态(Keyed State)



- 状态是根据输入流中定义的键 (key) 来维护和访问的，所以只能定义在按键分区流 (KeyedStream) 中，也就keyBy之后才可以使用
- 聚合算子必须在keyBy之后才能使用，就是因为聚合的结果是以Keyed State的形式保存的。
- `ValueState<T>`: 保存一个可以更新和检索的值 (如上所述，每个值都对应到当前的输入数据的 key，因此算子接收到的每个 key 都可能对应一个值)。这个值可以通过 `update(T)` 进行更新，通过 `T value()` 进行检索。
  - `ListState<T>`: 保存一个元素的列表。可以往这个列表中追加数据，并在当前的列表上进行检索。可以通过 `add(T)` 或者 `addAll(List<T>)` 进行添加元素，通过 `Iterable<T> get()` 获得整个列表。还可以通过 `update(List<T>)` 覆盖当前的列表。
  - `ReducingState<T>`: 保存一个单值，表示添加到状态的所有值的聚合。接口与 `ListState` 类似，但使用 `add(T)` 增加元素，会使用提供的 `ReduceFunction` 进行聚合。
  - `AggregatingState<IN, OUT>`: 保留一个单值，表示添加到状态的所有值的聚合。和 `ReducingState` 相反的是，聚合类型可能与 添加到状态的元素的类型不同。接口与 `ListState` 类似，但使用 `add(IN)` 添加的元素会用指定的 `AggregateFunction` 进行聚合。
  - `MapState<UK, UV>`: 维护了一个映射列表。你可以添加键值对到状态中，也可以获得反映当前所有映射的迭代器。使用 `put(UK, UV)` 或者 `putAll(Map<UK, UV>)` 添加映射。使用 `get(UK)` 检索特定 key。使用 `entries()`, `keys()` 和 `values()` 分别检索映射、键和值的可迭代视图。你还可以通过 `isEmpty()` 来判断是否包含任何键值对。

```
import org.apache.flink.api.common.functions.RichReduceFunction;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.io.File;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 */
```

```

* @Teacher:李毅大帝
* @Mail:863159469@qq.com
*/
public class Hello03StateKeyed {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(1);
        environment.enableCheckpointing(5000);
        environment.getCheckpointConfig().setCheckpointStorage("file:///"+
System.getProperty("user.dir") + File.separator + "ckpt");
        //获取数据源【水果:重量】
        DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
        //计算
        source.map(line -> {
            String[] split = line.split(":");
            return Tuple2.of(split[0], Integer.parseInt(split[1]));
        }, Types.TUPLE(Types.STRING, Types.INT))
        .keyBy(tuple2 -> tuple2.f0)
        .reduce(new YjxxtKeyedStateFunction())
        .print();
        //运行环境
        environment.execute();
    }
}

/**
 * 有钱可以为所欲为
 */
class YjxxtKeyedStateFunction extends RichReduceFunction<Tuple2<String,
Integer>> {
    //声明一个状态对象
    private ValueState<Tuple2<String, Integer>> valueState;

    @Override
    public Tuple2<String, Integer> reduce(Tuple2<String, Integer> value1,
Tuple2<String, Integer> value2) throws Exception {
        //开始计算
        value1.f1 = value1.f1 + value2.f1;

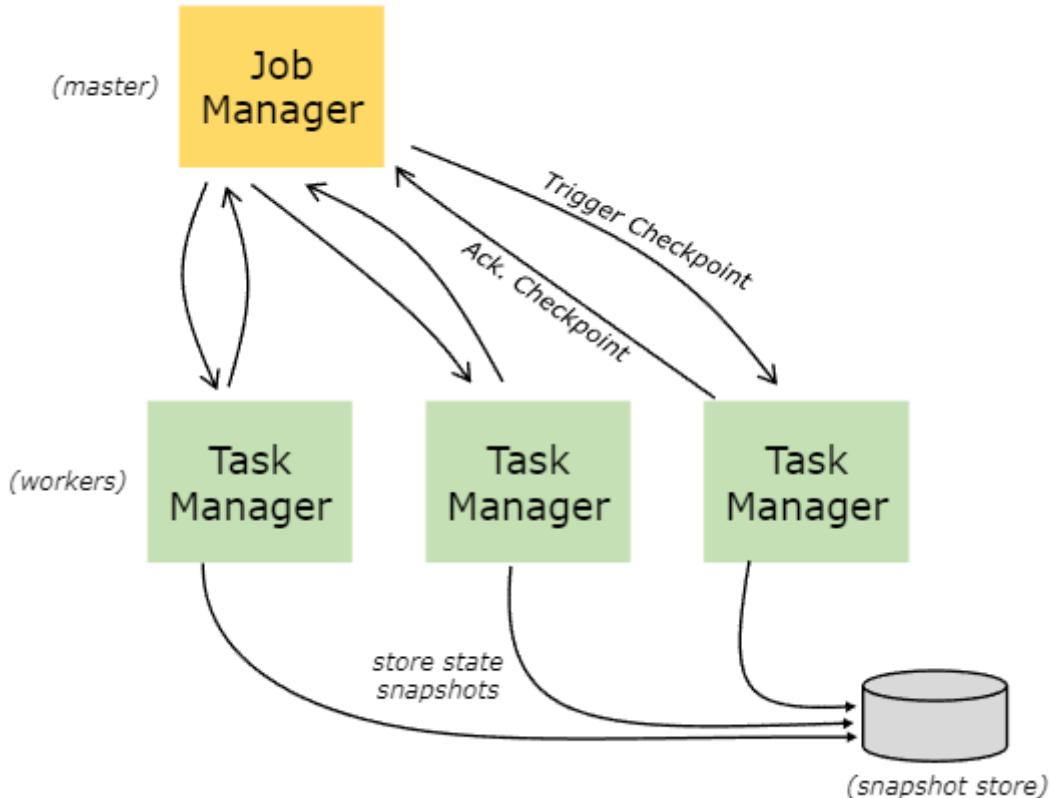
        //保存状态【自己动手丰衣足食】
        valueState.update(value1);

        return value1;
    }

    @Override
    public void open(Configuration parameters) throws Exception {
        //初始化状态对象
        ValueStateDescriptor<Tuple2<String, Integer>> descriptor = new
ValueStateDescriptor<>("reduceValueState", Types.TUPLE(Types.STRING,
Types.INT));
        this.valueState = getRuntimeContext().getstate(descriptor);
    }
}

```

## 1.4. 状态后端



- State Backends 的作用就是用来维护State的。
  - 有状态的流计算是Flink的一大特点，状态本质上是数据，数据是需要维护的，例如数据库就是维护数据的一种解决方案。
- 一个 State Backend 主要负责两件事：Local State Management(本地状态管理) 和 Remote State Checkpointing (远程状态备份) 。
  - Local State Management
    - State Management 的主要任务是确保状态的更新和访问。
    - State Backends 主要有两种形式的状态管理：
      - 直接将 State 以对象的形式存储到JVM的堆上面
      - 将 State 对象序列化后存储到 RocksDB 中
    - 第一种存储到JVM堆中，因为是在内存中读写，延迟会很低，但State的大小受限于内存的大小；第二种方式存储到State Backends上（本地磁盘上），读写较内存会慢一些，但不受内存大小的限制，同时因为state存储在磁盘上，可以减少应用程序对内存的占用。根据使用经验，对延迟不是特别敏感的应用，选择第二种方式较好，尤其是State比较大的情况下。
  - Remote State Checkpointing
    - Flink程序是分布式运行的，而State都是存储到各个节点上的，一旦TaskManager节点出现问题，就会导致State的丢失。
    - State Backend 提供了 State Checkpointing 的功能，将 TaskManager 本地的 State 的备份到远程的存储介质上，可以是分布式的存储系统或者数据库。
    - 不同的 State Backends 备份的方式不同，会有效率高低的区别。
  - 状态后端的主要作用包括在每一个TaskManager节点上存储和管理状态，将状态进行远程备份两个部分。

### 1.4.1. 存储方式

- flink-1.13 版及以前

区别	MemoryStateBackend	FsStateBackend	RocksDBStateBackend
存储方式	state:TaskManager内存 Checkpoint:JobManager内存	state:TaskManager内存 Checkpoint:外部文件系统(HDFS)	state:TaskManager上的RockDB(内存+磁盘) Checkpoint:外部文件系统(HDFS)
使用场景	本地测试	分钟级窗口聚合、join，生产环境使用	超大状态作业，天级窗口聚合，生产环境使用

- MemoryStateBackend

- 基于内存存储

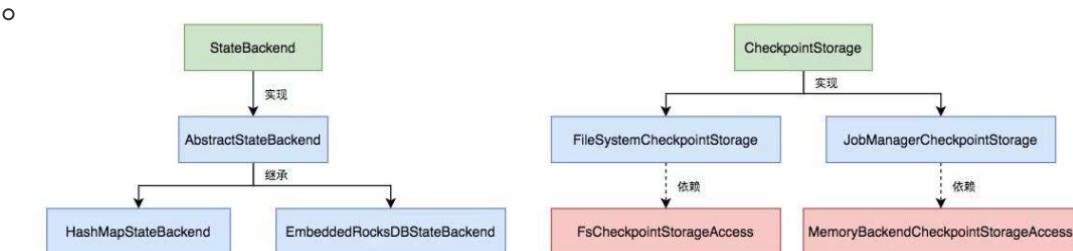
- FsStateBackend

- 基于文件存储
  - 本地路径，其格式为“file:///data/flink/checkpoints”
  - HDFS路径，其格式为“hdfs://nameservice/flink/checkpoints”

- RocksDBStateBackend

- 基于RocksDB存储
  - RocksDB 是一个 key/value 的内存存储系统，类似于HBase
  - 当写数据时会先写进write buffer(类似于HBase的memstore)，然后在flush到磁盘文件，
  - 当读取数据时会现在block cache(类似于HBase的block cache)，所以速度会很快。

- flink-1.13 版及以后



- HashMapStateBackend 【默认】

- Fsstatebackend 和 MemoryStatebackend 整合成了 HashMapStateBackend
  - 存储管理状态类似于管理一个java堆中的对象，key/value的状态和窗口操作都会在一个hash table中进行存储状态。
  - 默认情况下，每一个状态最大为 5 MB。可以通过 MemoryStateBackend 的构造函数增加最大大小。
  - 内存空间不够时，也会溢出一部分数据到本地磁盘文件；
  - 可以支撑大规模的状态数据,只不过在状态数据规模超出内存空间时，读写效率就会明显降低

- EmbeddedRocksDBStateBackend

- 该状态后端存储管理状态在RocksDB数据库中，该状态存储在本地taskManager的磁盘目录。
  - 该状态后端按照指定的类型序列化后变成字节数组将数据存储在磁盘，按照key的字典序排列
  - RocksDB 的每个 key 和 value 的最大大小为  $2^{31}$  字节。这是因为 RocksDB 的 JNI API 是基于 byte[] 的。
  - Rockdb 中的数据，有内存缓存的部分，也有磁盘文件的部分；

- Rockdb 的磁盘文件数据读写速度相对还是比较快的，所以在支持超大规模状态数据时，数据的读写效率不会有太大的降低
- 代码实现

```
//设置内存状态后端
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new HashMapStateBackend());

//设置RockSDB状态后端
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new EmbeddedRocksDBStateBackend());

//设置checkpoint内存存储
env.getCheckpointConfig().setCheckpointStorage(new
JobManagerCheckpointStorage());
//设置checkpoint文件存储
env.getCheckpointConfig().setCheckpointStorage("file:///checkpoint-
dir");
env.getCheckpointConfig().setCheckpointStorage("hdfs://namenode/fli
nk/checkpoints");
```

### 1.4.2. 方式选择

- 从性能维度选择
  - hash后端保存状态在内存中，所以拥有完美的性能，但是恰因为状态留在内存，所以在扩展性方面取决于当前集群的内存量。
  - DB后端将状态存储在磁盘里，扩展性方面暂时不存在任何性能瓶颈，但是在保存状态和取用状态的时候需要通过序列化和反序列化，这样就导致了性能方面的劣势，可能在处理过程中达不到hash后端性能的平均水平。
- 从扩展维度选择
  - hash后端保存状态在内存中，虽然也可以存储到硬盘，但是扩展后性能会有影响
  - DB后端将状态存储在磁盘里，所以扩展后性能基本不会有影响
- HashMapStateBackend 和 EmBeddedRocksDBStateBackend 所生成的快照文件也统一了格式
  - flink 1.13版本加入了一个新功能，可以保存二进制的保存点，然后程序从该点恢复，恢复的时候可以任意选择使用hash恢复还是DB恢复，这样一来就可以随意切换状态后端。
  - 如果你想切换，那么先手动生成一个保存点，然后恢复的时候选择你想要的后端方式即可。

```
package com.yjxxt.flink;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.state.ListState;
import org.apache.flink.api.common.state.ListStateDescriptor;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackend;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/***
```

```

* @Description :
* @School:优极限学堂
* @Official-Website: http://www.yjxxt.com
* @Teacher:李毅大帝
* @Mail:863159469@qq.com
*/
public class Hello04StateBackend {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(2);
        environment.enableCheckpointing(5000);
        //本地状态维护
        environment.setStateBackend(new EmbeddedRocksDBStateBackend());
        //远程状态备份

        environment.getCheckpointConfig().setCheckpointStorage("hdfs://node02:8020/
flink/checkpoints");
        //获取Source
        DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
        //转换数据
        source.map(new YjxxtStateBackendFunction()).print();
        //运行环境
        environment.execute();
    }
}

class YjxxtStateBackendFunction implements MapFunction<String, String>,
CheckpointedFunction {

    //声明变量计数器
    private int count;
    //状态对象
    private ListState<Integer> listState;

    @Override
    public String map(String value) throws Exception {
        //计数器累加
        count++;
        return "[" + value.toUpperCase() + "][" + count + "]";
    }

    @Override
    public void snapshotState(FunctionSnapshotContext
functionsnapshotContext) throws Exception {
        //清空并更新数据
        listState.clear();
        listState.add(count);
    }

    @Override
    public void initializeState(FunctionInitializationContext context)
throws Exception {
        //创建描述器并创建对象
        ListStateDescriptor<Integer> descriptor = new ListStateDescriptor<>(
"ListState", Types.INT);

```

```

        this.liststate =
context.getOperatorStateStore().getListState(descriptor);
    }
}

```

## 1.5. 状态TTL

### 1.5.1. 基本概念

- Flink 可以对状态数据进行存活时长管理，即“新陈代谢”；
- 淘汰的机制主要是基于存活时间(Time To Live)；存活时长的计时器可以在数据被读、写时重置；
- TTL 存活管理粒度是到元素级的（如 liststate 中的每个元素，mapstate 中的每个 entry）

### 1.5.2. 相关参数

- TTI 的相关配置参数及其内含的机制，全部封装在 StateTtlConfig类中
- StateTtlConfig各参数详解
  - setTtl
    - 表示状态的过期时间，是一个 org.apache.flink.api.common.time.Time 对象。
    - 一旦设置了 TTL，那么如果上次访问的时间戳 + TTL 超过了当前时间，则表明状态过期了
  - setUpdateType
    - 表示状态时间戳的更新的时机，是一个 Enum 对象。
    - org.apache.flink.api.common.state.StateTtlConfig.UpdateType
  - setStateVisibility
    - 表示对已过期但还未被清理掉的状态如何处理，也是 Enum 对象。
    - org.apache.flink.api.common.state.StateTtlConfig.StateVisibility
  - ttlTimeCharacteristic
    - 表示 State TTL 功能所适用的时间模式，仍然是 Enum 对象。
    - org.apache.flink.api.common.state.StateTtlConfig.TtlTimeCharacteristic

策略类型	描述
StateTtlConfig.UpdateType.Disabled	禁用TTL，永不过期
StateTtlConfig.UpdateType.OnCreateAndWrite	每次写操作都会更新State的最后访问时间
StateTtlConfig.UpdateType.OnReadAndWrite	每次读写操作都会跟新State的最后访问时间

- setStateVisibility
  - 表示对已过期但还未被清理掉的状态如何处理，也是 Enum 对象。
  - org.apache.flink.api.common.state.StateTtlConfig.StateVisibility

策略类型	描述
StateTtlConfig.StateVisibility.NeverReturnExpired	永不返回过期状态
StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp	可以返回过期但尚未被清理的状态值

- ttlTimeCharacteristic
  - 表示 State TTL 功能所适用的时间模式，仍然是 Enum 对象。
  - org.apache.flink.api.common.state.StateTtlConfig.TtlTimeCharacteristic

- ProcessingTime
- cleanupStrategies
  - 表示过期对象的清理策略，目前来说有三种 Enum 值。
  - org.apache.flink.api.common.state.StateTtlConfig.CleanupStrategies.Strategies

策略类型	描述
FULL_STATE_SCAN_SNAPSHOT	默认情况下，过期值只有在显式读出时才会被删除 例如通过调用 ValueState.value() 方法。
INCREMENTAL_CLEANUP	增量地触发对某些状态项的清理。触发器可以是来自每个状态访问或/和每个记录处理的回调。 触发器可以是来自每个状态访问或/和每个记录处理的回调。
ROCKSDB_COMPACTION_FILTER;	RocksDB会定期使用异步压缩来合并状态的更新和减少储存。 Flink压缩过滤器使用TTL检查状态的过期时间戳，并排除过期值。默认情况下是关闭该特性的。 对于RocksDB进行状态管理首先要做的就是要激活，通过Flink配置文件配置 state.backend.rocksdb.ttl.compaction.filter.enabled，或者对于一个Flink job来说如果一个自定义的RocksDB 状态管理被创建那么它可以调用 RocksDBStateBackend::enableTtlCompactionFilter。

- 代码实现

```

○ StateTtlConfig.newBuilder(Time.seconds(60))
    // 配置数据存活时间为4s（覆盖builder构造传入的1s）
    .setTtl(Time.milliseconds(4000))
    // 当插入、更新时候，该数据的ttl计时重置
    .updateTtlOnCreateAndWrite()
    // 当读取、更新时候，该数据的ttl计时重置
    .updateTtlOnReadAndWrite()
    // 不允许返回已经过期但是还没清理的数据

    .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
    // 允许返回已经过期但是还没清理的数据

    .setStateVisibility(StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp)
    // ttl的时间语义：设置为处理时间

    .setTtlTimeCharacteristic(StateTtlConfig.TtlTimeCharacteristic.ProcessingTime)
    // ttl的时间语义：设置为处理时间
    .useProcessingTime()
    // 增量清理（每一条状态数据被访问，会驱动过期检查以及清除）
    .cleanupIncrementally(1000, true)
    // 全量快照清理策略（ck时候，保存到快照文件的值包含未过期的状态数据，并不会清理算子状态数据）
    .cleanupFullSnapshot()
    // compact 过程中清理过期的状态数据
    .cleanupInRocksdbCompactFilter(1000)
    // 禁用默认后台清理策略
    .disableCleanupInBackground()
    .build();

```

## 2. Flink 窗口联结

- 对于两条流的合并，很多情况我们并不是简单地将所有数据放在一起，而是希望根据某个字段的值将它们联结起来，“配对”去做处理。

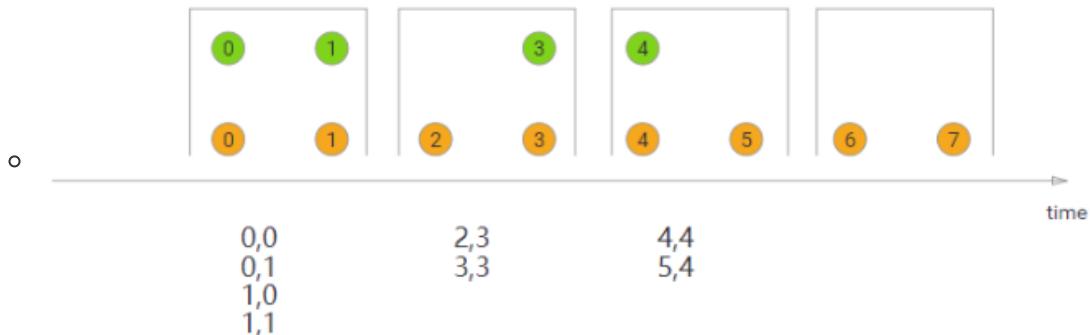
### 2.1. Join

- join都是利用window的机制，即按照指定字段和（滚动/滑动/会话）窗口进行inner join
- 先将数据缓存在Window State中，当窗口触发计算时，执行join操作；
- 按照窗口的操作和类型可以分为：
  - Tumbling Window Join、Sliding Window Join、Session Widnow Join。

• <b>Window Join</b> DataStream,DataStream → DataStream	Join two data streams on a given key and a common window. <pre>dataStream.join(otherStream)     .where(0).equalTo(1)     .window(TumblingEventTimeWindows.of(Time.seconds(3)))     .apply { ... }</pre>
--	--

#### 2.1.1. Tumbling Window Join

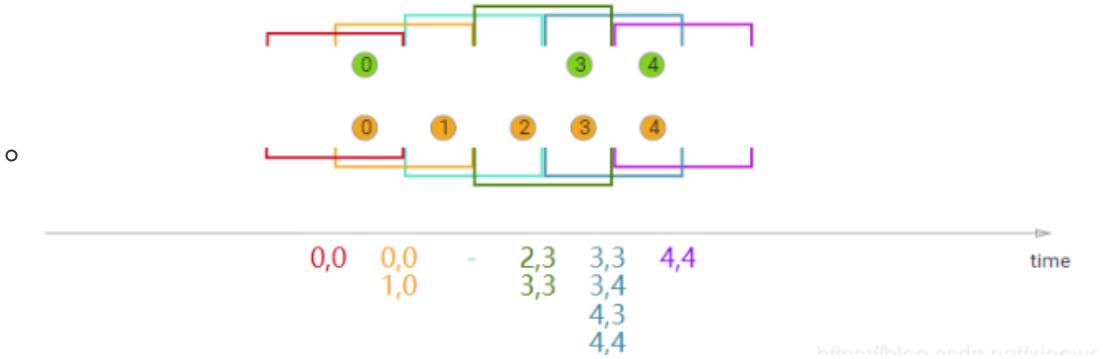
- 执行翻滚窗口联接时，具有公共键和公共翻滚窗口的所有元素将作为成对组合联接，并传递给JoinFunction或FlatJoinFunction。
- 因为它的行为类似于内部连接，所以一个流中的元素在其滚动窗口中没有来自另一个流的元素，因此不会被发射！
- 案例：



- 我们定义了一个大小为2毫秒的翻滚窗口，结果窗口的形式为[0,1]、[2,3]、...
- 该图显示了每个窗口中所有元素的成对组合，这些元素将传递给JoinFunction。
- 注意，在翻滚窗口[6,7]中没有发射任何东西，因为绿色流中不存在与橙色元素⑥和⑦结合的元素。

#### 2.1.2. Sliding Window Join

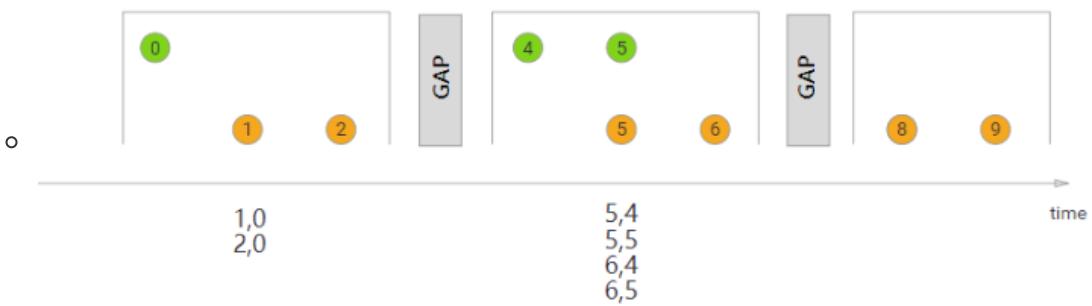
- 在执行滑动窗口联接时，具有公共键和公共滑动窗口的所有元素将作为成对组合联接，并传递给JoinFunction或FlatJoinFunction。
- 在当前滑动窗口中，一个流的元素没有来自另一个流的元素，则不会发射！
- 请注意，某些元素可能会连接到一个滑动窗口中，但不会连接到另一个滑动窗口中！
- 案例：



- 我们使用大小为2毫秒的滑动窗口，并将其滑动1毫秒，从而产生滑动窗口[-1, 0], [0,1], [1,2], [2,3]....
- x轴下方的连接元素是传递给每个滑动窗口的JoinFunction的元素。在这里，您还可以看到，例如，在窗口[2,3]中，橙色②与绿色③连接，但在窗口[1,2]中没有与任何对象连接。

### 2.1.3. Session Window Join

- 在执行会话窗口联接时，具有相同键（当“组合”时满足会话条件）的所有元素以成对组合方式联接，并传递给JoinFunction或FlatJoinFunction。
- 同样，这执行一个内部连接，所以如果有一个会话窗口只包含来自一个流的元素，则不会发出任何输出！
- 案例：



- 我们定义了一个会话窗口连接，其中每个会话被至少1ms的间隔分割。
- 有三个会话，在前两个会话中，来自两个流的连接元素被传递给JoinFunction。在第三个会话中，绿色流中没有元素，所以⑧和⑨没有连接！

- 代码实现

```

import com.yjxxt.util.kafkaUtil;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.JoinFunction;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.time.Duration;

/**

```

```

* @Description :
* @School:优极限学堂
* @Official-website: http://www.yjxxt.com
* @Teacher:李毅大帝
* @Mail:863159469@qq.com
*/
public class Hello06Join {
    public static void main(String[] args) throws Exception {

        //创建一个线程生成数据
        new Thread(() -> {
            for (int i = 100; i < 200; i++) {
                //生成一个商品ID
                String goodId =
                    RandomStringUtils.randomAlphabetic(16).toLowerCase();
                //发送goodInfo数据 [id:info:ts]
                KafkaUtil.sendMsg("t_goodinfo", goodId + ":info" + i +
                    ":" + System.currentTimeMillis());
                //创建goodPrice数据[id:price:ts]
                KafkaUtil.sendMsg("t_goodprice", goodId + ":" + i + ":" +
                    (System.currentTimeMillis() - 5000));
                //让线程休眠一下
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        //运行环境
        StreamExecutionEnvironment environment =
            StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(2);
        //获取数据源
        DataStreamSource<String> goodInfoSource =
            environment.fromSource(KafkaUtil.getKafkaSource("t_goodinfo", "liyi"),
                WatermarkStrategy.noWatermarks(), "Kafka Source Info");
        DataStreamSource<String> goodPriceSource =
            environment.fromSource(KafkaUtil.getKafkaSource("t_goodprice", "liyi"),
                WatermarkStrategy.noWatermarks(), "Kafka Source Price");
        //添加水位线
        SingleOutputStreamOperator<Tuple3<String, String, Long>>
            infoStream = goodInfoSource.map(record -> {
                String[] split = record.split(":");
                return Tuple3.of(split[0], split[1],
                    Long.parseLong(split[2]));
            }, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
            .assignTimestampsAndWatermarks(WatermarkStrategy.
                <Tuple3<String, String,
                Long>>forBoundedOutOfOrderDiversity(Duration.ofSeconds(3))
                    .withTimestampAssigner((element, recordTime) ->
            {
                return element.f2;
            }));
        SingleOutputStreamOperator<Tuple3<String, String, Long>>
            priceStream = goodPriceSource.map(record -> {
                String[] split = record.split(":");

```

```

        return Tuple3.of(split[0], split[1],
Long.parseLong(split[2]));
    }, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(WatermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderDurations(Duration.ofSeconds(3))
.withTimestampAssigner((element, recordTime) ->
{
    return element.f2;
})));
//开始进行流的Join
// infostream.join(priceStream)
//     .where(i -> i.f0)
//     .equalTo(p -> p.f0)
//
.window(TumblingEventTimeWindows.of(Time.seconds(10)))
//     .apply(new JoinFunction<Tuple3<String, String,
Long>, Tuple3<String, String, Long>, String>() {
//         @Override
//         public String join(Tuple3<String, String, Long>
info, Tuple3<String, String, Long> price) throws Exception {
//             return "[" + info + "][" + price + "]";
//         }
//     }).print("TumblingEventTimeWindows--");

//开始进行流的Join
infostream.join(priceStream)
.where(i -> i.f0)
.equalTo(p -> p.f0)
.window(SlidingEventTimeWindows.of(Time.seconds(10),
Time.seconds(3)))
//     .apply(new JoinFunction<Tuple3<String, String, Long>,
Tuple3<String, String, Long>, String>() {
//         @Override
//         public String join(Tuple3<String, String, Long>
info, Tuple3<String, String, Long> price) throws Exception {
//             return "[" + info + "][" + price + "]";
//         }
//     }).print("SlidingEventTimeWindows--");

//运行环境
environment.execute();
}

}
}

```

## 2.2. CoGroup

- CoGroup: 除了输出匹配的元素对以外，未能匹配的元素也会输出。

- Window CoGroup  
DataStream,DataStream →  
DataStream

Cogroups two data streams on a given key and a common window.

```

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

```

- 用于DataStream时返回是CoGroupedStreams，用于DataSet时返回是CoGroupOperatorSets
- 代码实现

```

    ○ import com.yjxxt.util.kafkaUtil;
    import org.apache.commons.lang3.RandomStringUtils;
    import org.apache.flink.api.common.eventtime.WatermarkStrategy;
    import org.apache.flink.api.common.functions.CoGroupFunction;
    import org.apache.flink.api.common.typeinfo.Types;
    import org.apache.flink.api.java.tuple.Tuple3;
    import org.apache.flink.streaming.api.datastream.DataStreamSource;
    import
    org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
    import
    org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
    import
    org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWin
    dows;
    import org.apache.flink.streaming.api.windowing.time.Time;
    import org.apache.flink.util.Collector;

    import java.time.Duration;

    /**
     * @Description :
     * @School:优极限学堂
     * @Official-website: http://www.yjxxt.com
     * @Teacher:李毅大帝
     * @Mail:863159469@qq.com
     */
    public class Hello07CoGroup {
        public static void main(String[] args) throws Exception {

            //创建一个线程生成数据
            new Thread(() -> {
                for (int i = 100; i < 200; i++) {
                    //生成一个商品ID
                    String goodId =
                    RandomStringUtils.randomAlphabetic(16).toLowerCase();
                    //发送goodInfo数据 [id:info:ts]
                    KafkaUtil.sendMsg("t_goodinfo", goodId + ":info" + i +
                    ":" + System.currentTimeMillis());
                    if (i % 5 != 0) {
                        //创建goodPrice数据[id:price:ts]
                        KafkaUtil.sendMsg("t_goodprice", goodId + ":" + i +
                        ":" + System.currentTimeMillis());
                    }
                    //让线程休眠一下
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }).start();

            //运行环境
        }
    }

```

```

        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(2);
        //获取数据源
        DataStreamSource<String> goodInfoSource =
environment.fromSource(KafkaUtil.getKafkaSource("t_goodinfo", "liyi"),
watermarkStrategy.noWatermarks(), "Kafka Source Info");
        DataStreamSource<String> goodPriceSource =
environment.fromSource(KafkaUtil.getKafkaSource("t_goodprice", "liyi"),
watermarkStrategy.noWatermarks(), "Kafka Source Price");
        //添加水位线
        SingleOutputStreamOperator<Tuple3<String, String, Long>>
infoStream = goodInfoSource.map(record -> {
            String[] split = record.split(":");
            return Tuple3.of(split[0], split[1],
Long.parseLong(split[2]));
        }, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(watermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderliness(Duration.ofSeconds(3))
.withTimestampAssigner((element, recordTime) ->
{
            return element.f2;
        }));
        SingleOutputStreamOperator<Tuple3<String, String, Long>>
priceStream = goodPriceSource.map(record -> {
            String[] split = record.split(":");
            return Tuple3.of(split[0], split[1],
Long.parseLong(split[2]));
        }, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(watermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderliness(Duration.ofSeconds(3))
.withTimestampAssigner((element, recordTime) ->
{
            return element.f2;
        }));
        //开始进行流的Join
        infoStream.coGroup(priceStream)
.where(i -> i.f0)
.equalTo(p -> p.f0)
.window(TumblingEventTimeWindows.of(Time.seconds(10)))
.apply(new CoGroupFunction<Tuple3<String, String,
Long>, Tuple3<String, String, Long>, String>() {
        @Override
        public void coGroup(Iterable<Tuple3<String, String,
Long>> info, Iterable<Tuple3<String, String, Long>> price,
Collector<String> out) throws Exception {
            String s =
RandomStringUtils.randomAlphabetic(8);
            //收集Info
            for (Tuple3<String, String, Long> tuple3 :
info) {
                out.collect(s + "--" + tuple3.toString());
            }
            //收集Price
            for (Tuple3<String, String, Long> tuple3 :
price) {

```

```

        out.collect(s + " -- " + tuple3.toString());
    }
}
}).print("CoGroup--").setParallelism(1);

//运行环境
environment.execute();
}
}

```

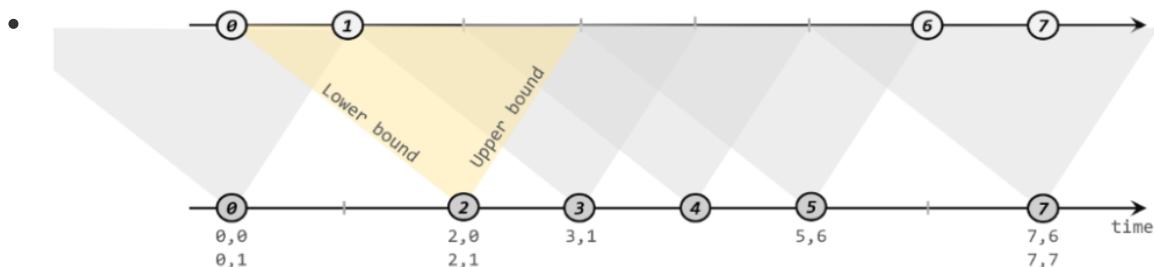
## 2.3. Interval Join

### 2.3.1. 业务需求

- 在有些场景下，我们要处理的时间间隔可能并不是固定的。
  - 比如，在交易系统中，需要实时地对每一笔交易进行核验，保证两个账户转入转出数额相等，也就是所谓的“实时对账”。
  - 两次转账的数据可能写入了不同的日志流，它们的时间戳应该相差不大，所以我们可以考虑只统计一段时间内是否有出账入账的数据匹配。
  - 显然不应该用滚动窗口或滑动窗口来处理——因为匹配的两个数据有可能刚好“卡在”窗口边缘两侧，于是窗口内就都没有匹配了；会话窗口虽然时间不固定，但也明显不适合这个场景。
- 为了应对基于时间的窗口联结已经无能为力的需求，Flink提供了一种叫作“间隔联结”（interval join）的合流操作。
- 间隔联结的思路就是针对一条流的每个数据，开辟出其时间戳前后的一段时间间隔，看这期间是否有来自另一条流的数据匹配。

### 2.3.2. 实现原理

- 间隔联结具体的定义方式是，我们给定两个时间点，分别叫作间隔的“上界”（upperBound）和“下界”（lowerBound）
- 对于一条流（不妨叫作A）中的任意一个数据元素a，就可以开辟一段时间间隔，把这段时间作为可以匹配另一条流数据的“窗口”范围。
  - 以a的时间戳为中心，下至下界点、上至上界点的一个闭区间
  - $[a.timestamp + lowerBound, a.timestamp + upperBound]$
- 对于另一条流（不妨叫B）中的数据元素b，如果它的时间戳落在了这个区间范围内，a和b就可以成功配对，进而进行计算输出结果。所以匹配的条件为：
  - $a.timestamp + lowerBound \leq b.timestamp \leq a.timestamp + upperBound$
- 间隔联结的两条流A和B，也必须基于相同的key；下界lowerBound应该小于等于上界upperBound，两者都可正可负；间隔联结目前只支持事件时间语义。



### 2.3.3. 代码实现

- 间隔联结在代码中，是基于KeyedStream的联结（join）操作。
- DataStream在keyBy得到KeyedStream之后，可以调用intervalJoin()来合并两条流，传入的参数同样是一个KeyedStream，两者的key类型应该一致；得到的是一个IntervalJoin类型。后续的操作同样是完全固定的：先通过between()方法指定间隔的上下界，再调用process()方法，定义对匹配数据对的处理操作。调用process()需要传入一个处理函数，这是处理函数家族的最后一员：“处理联结函数”ProcessJoinFunction。

```
• import com.yjxxt.util.KafkaUtil;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

import java.time.Duration;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello08IntervalJoin {
    public static void main(String[] args) throws Exception {

        //创建2个线程生成数据
        new Thread(() -> {
            for (int i = 100; i < 200; i++) {
                //发送goodInfo数据 [id:info:ts]
                KafkaUtil.sendMsg("t_goodinfo", i + ":info" + i + ":" +
System.currentTimeMillis());
                //让线程休眠一下
                try {
                    Thread.sleep((int) (Math.random() * 3000));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        new Thread(() -> {
            for (int i = 100; i < 200; i++) {
                //创建goodPrice数据[id:price:ts]
                KafkaUtil.sendMsg("t_goodprice", i + ":" + i + ":" +
System.currentTimeMillis());
                //让线程休眠一下
                try {
                    Thread.sleep((int) (Math.random() * 3000));
                } catch (InterruptedException e) {
```

```

        e.printStackTrace();
    }
}
}).start();

//运行环境
StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
environment.setParallelism(2);
//获取数据源
DataStreamSource<String> goodInfoSource =
environment.fromSource(KafkaUtil.getKafkaSource("t_goodinfo", "liyi"),
WatermarkStrategy.noWatermarks(), "Kafka Source Info");
DataStreamSource<String> goodPricesource =
environment.fromSource(KafkaUtil.getKafkaSource("t_goodprice", "liyi"),
WatermarkStrategy.noWatermarks(), "Kafka Source Price");
//添加水位线
SingleOutputStreamOperator<Tuple3<String, String, Long>> infostream
= goodInfoSource.map(record -> {
    String[] split = record.split(":");
    return Tuple3.of(split[0], split[1],
    Long.parseLong(split[2]));
}, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(WatermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderliness(Duration.ofSeconds(3))
.withTimestampAssigner((element, recordTime) -> {
    return element.f2;
}));;
SingleOutputStreamOperator<Tuple3<String, String, Long>> priceStream
= goodPricesource.map(record -> {
    String[] split = record.split(":");
    return Tuple3.of(split[0], split[1],
    Long.parseLong(split[2]));
}, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(WatermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderliness(Duration.ofSeconds(3))
.withTimestampAssigner((element, recordTime) -> {
    return element.f2;
}));;
//开始进行流的Join
infostream.keyBy(info -> info.f0)
.intervalJoin(priceStream.keyBy(price -> price.f0))
.between(Time.seconds(-2), Time.seconds(2))
.process(new ProcessJoinFunction<Tuple3<String, String,
Long>, Tuple3<String, String, Long>, String>() {
    @Override
    public void processElement(Tuple3<String, String, Long>
info, Tuple3<String, String, Long> price, ProcessJoinFunction<Tuple3<String,
String, Long>, Tuple3<String, String, Long>, String>.Context context,
Collector<String> collector) throws Exception {
        collector.collect("[ " + info + " ] [ " + price + " ] [ " +
context + " ]");
    }
})
.print("intervalJoin--").setParallelism(2);

```

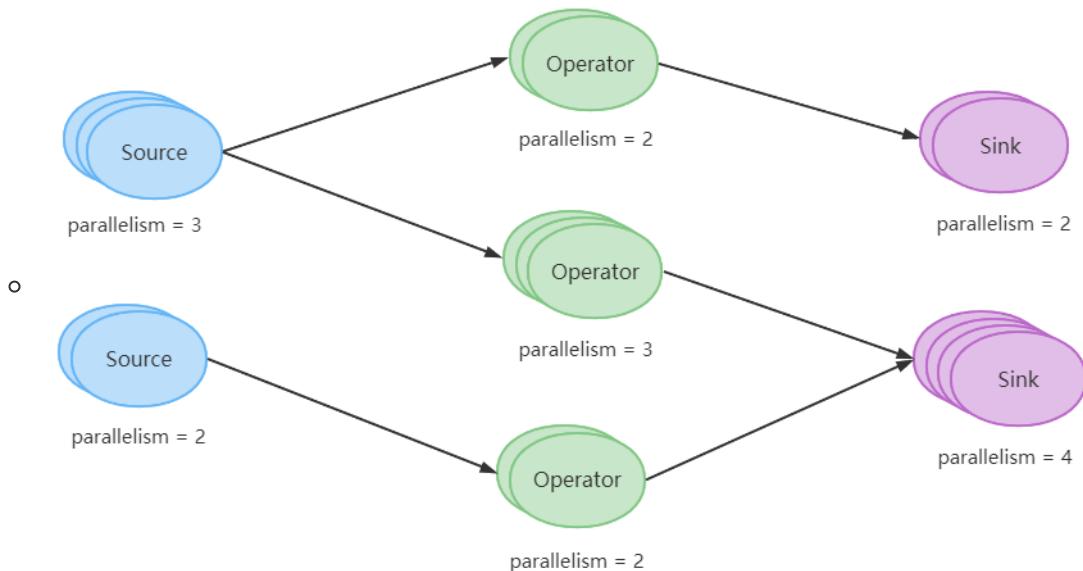
```
//运行环境  
environment.execute();  
}  
}
```

## 3. Flink 容错机制

- Flink 是一个 stateful (带状态) 的数据处理系统；系统在处理数据的过程中，各算子所记录的状态会随着数据的处理而不断变化。
- 一旦系统崩溃，需要重启后能够恢复出崩溃前的状态才能进行数据的继续处理。
- 因此，必须要一种机制能对系统内的各种状态进行持久化容错。
- Flink-EOS:Exactly-Once Semantics
  - 指端到端的一致性，从数据读取、引擎计算、写入外部存储的整个过程中，即使机器或软件出现故障，都确保数据仅处理一次，不会重复、也不会丢失。
  - 一条（或者一批）数据，从注入系统、中间处理、到输出结果的整个流程中，要么每个环节都处理成功，要么失败回滚
  - Flink 在目前的各类分布式计算引擎中，对 EOS 的支持是最完善的

### 3.1. 数据处理语义

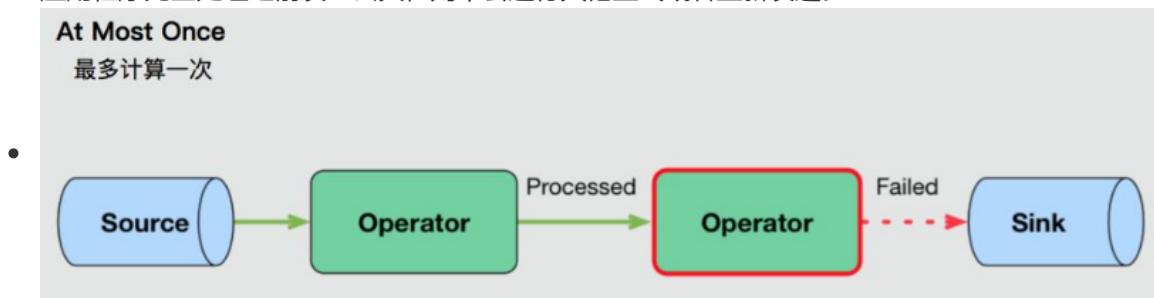
- 对于批处理，fault-tolerant (容错性) 很容易做，失败只需要replay，就可以完美做到容错。
- 对于流处理，数据流本身是动态，没有所谓的开始或结束，虽然可以replay buffer的部分数据，但fault-tolerant做起来会复杂的多。
- 有向无环图
  - 流处理可以简单地描述为是对无界数据或事件的连续处理。流或事件处理应用程序可以或多或少地被描述为有向图，并且通常被描述为有向无环图 (DAG) 。
  - 每个边表示数据或事件流，每个顶点表示运算符，会使用程序中定义的逻辑处理来自相邻边的数据或事件。
  - 有两种特殊类型的顶点，通常称为 sources 和 sinks。sources读取外部数据/事件到应用程序中，而 sinks 通常会收集应用程序生成的结果。
- 分布式下的有向无环图
  - 分布式情况下是由多个Source(读取数据)节点、多个Operator(数据处理)节点、多个Sink(输出)节点构成
  - 每个节点的并行数可以有差异，且每个节点都有可能发生故障
  - 对于数据正确性最重要的一点，就是当发生故障时，是怎样容错与恢复的。



- 流处理引擎通常为应用程序提供了三种数据处理语义：最多一次、至少一次和精确一次。
  - (一致性由弱到强): At most once < At least once < Exactly once < End to End Exactly once

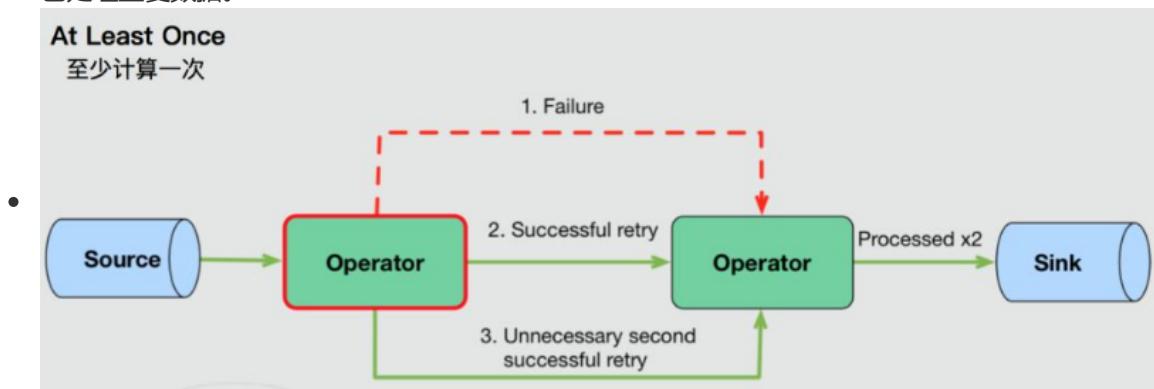
### 3.1.1. At-most-once

- 有可能会有数据丢失。**
- 这本质上是简单的恢复方式，也就是直接从失败处的下个数据开始恢复程序，之前的失败数据处理就不管了。可以保证数据或事件最多由应用程序中的所有算子处理一次。这意味着如果数据在被流应用程序完全处理之前发生丢失，则不会进行其他重试或者重新发送。



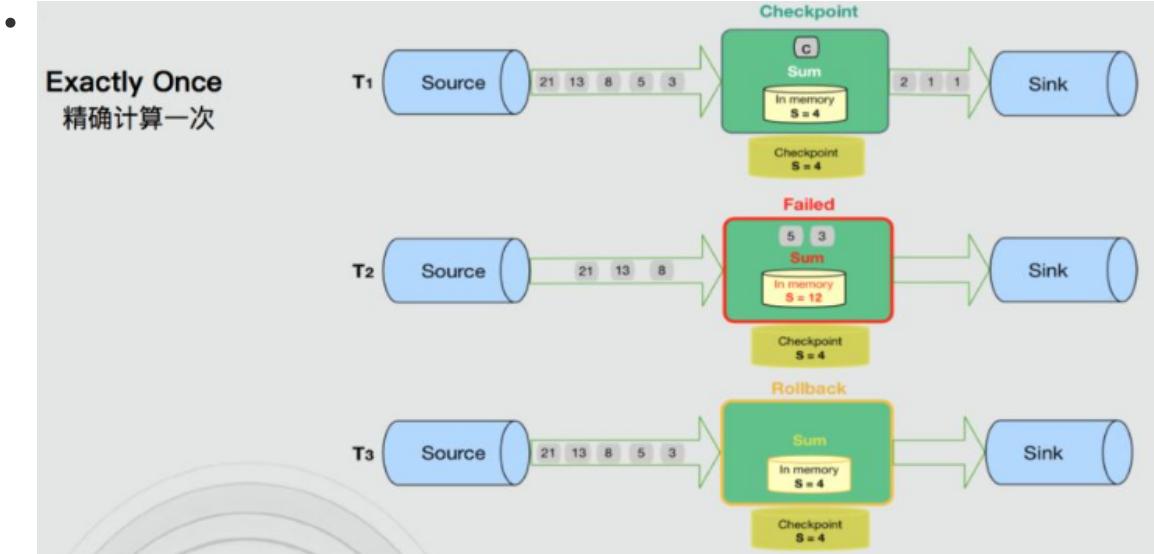
### 3.1.2. At-least-once

- 有可能重复处理数据。**
- 应用程序中的所有算子都保证数据或事件至少被处理一次。这通常意味着如果事件在流应用程序完全处理之前丢失，则将从源头重放或重新传输事件。然而，由于事件是可以被重传的，因此一个事件有时会被处理多次(至少一次)，至于有没有重复数据，不会关心，所以这种场景需要人工干预自己处理重复数据。



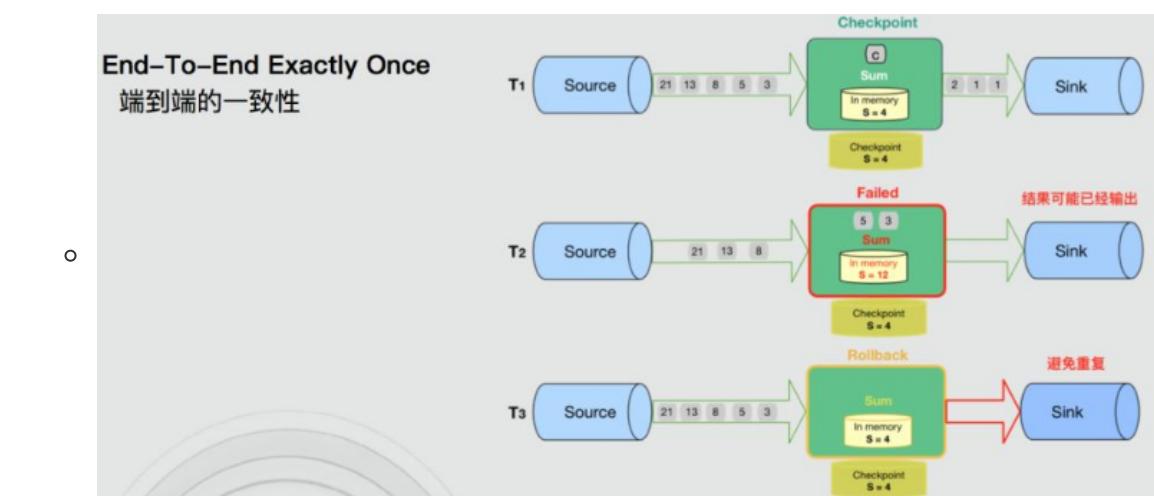
### 3.1.3. Exactly-once

- 每一条消息只被流处理系统处理一次。
- 即使是在各种故障的情况下，流应用程序中的所有算子都保证事件只会被『精确一次』的处理。
- Flink实现『精确一次』的分布式快照/状态检查点方法受到 Chandy-Lamport 分布式快照算法的启发。
  - 流应用程序中每个算子的所有状态都会定期做 checkpoint。
  - 如果是在系统中的任何地方发生失败，每个算子的所有状态都回滚到最新的全局一致 checkpoint 点。
  - 在回滚期间，将暂停所有处理。源也会重置为与最近 checkpoint 相对应的正确偏移量。
  - 整个流应用程序基本上是回到最近一次的一致状态，然后程序从该状态重新启动。

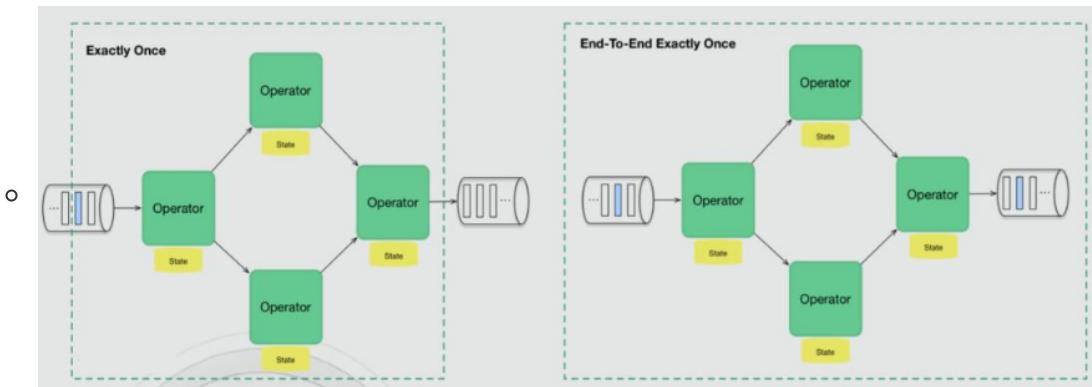


### 3.1.4. End-to-End Exactly-Once

- 端到端的精确一次。
- Flink 在1.4.0 版本引入『Exactly-Once』并号称支持『End-to-End Exactly-Once』“端到端的精确一次”语义。
- 它指的是 Flink 应用从 Source 端开始到 Sink 端结束，数据必须经过的起始点和结束点。



- Exactly-Once和End-to-End Exactly-Once的对比：
  - Exactly-Once：保证所有记录仅影响内部状态一次
  - End-to-End Exactly-Once：保证所有记录仅影响内部和外部状态一次

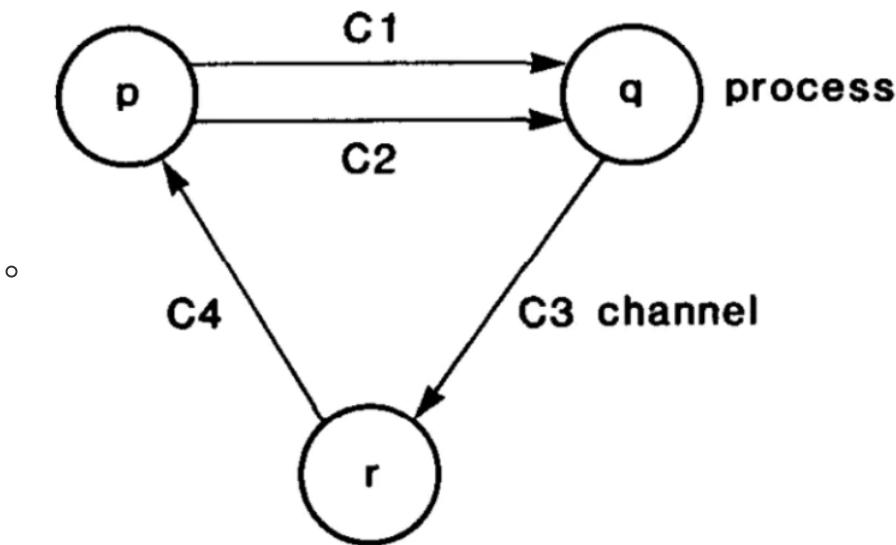


## 3.2. CheckPoint

- checkpoint是flink中的一种容错机制，使得任务失败的时候可以进行重启而不丢失之前的一些信息（需要数据源支持重发机制）。

### 3.2.1. Chandy-Lamport

- 分布式快照 (Distributed Snapshot)
  - 分布式快照：特定时间点记录下来的分布式系统的全局状态 (global state)。
  - 分布式快照主要用途：故障恢复（即检查点）、死锁检测、垃圾收集等。
- 数据模型
  - 为了定义分布式的全局状态，我们先将分布式的系统简化成有限个进程和进程之间的 channel 组成，也就是一个有向图：节点是进程，边是 channel。
  - 因为是分布式的系统，也就是说，这些进程是运行在不同的物理机器上的。
  - 那么一个分布式的全局状态就是有进程的状态和 channel 中的 message 组成，这个也是分布式的快照算法需要记录的。
  - global state要包含所有进程的状态以及所有channel的状态。

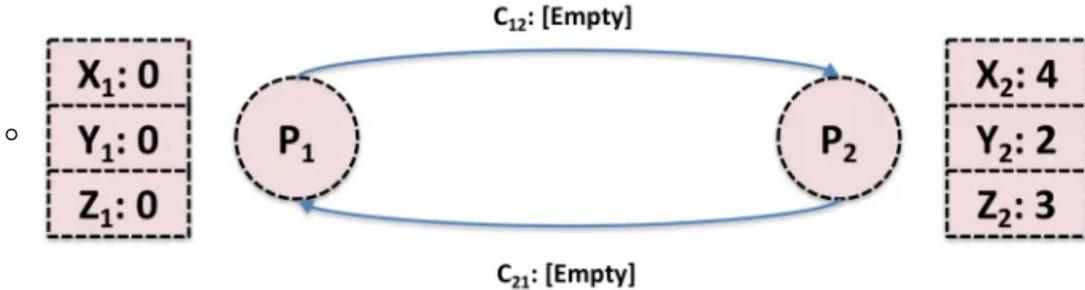


- Chandy-Lamport简介
  - Chandy-Lamport 算法以两个作者的名字命名，其中 Lamport 就是分布式系统领域无人不晓的 Leslie Lamport，著名的一致性算法 Paxos 的作者。
  - 算法的论文于 1985 年发表，*Distributed Snapshots: Determining Global States of a Distributed System*

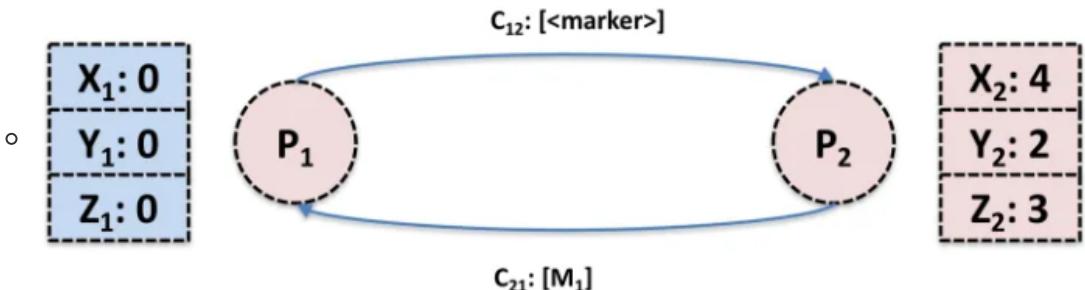
- The distributed snapshot algorithm described here came about when I visited Chandy, who was then at the University of Texas in Austin. He posed the problem to me over dinner, but we had both had too much wine to think about it right then. The next morning, in the shower, I came up with the solution. When I arrived at Chandy's office, he was waiting for me with the same solution. I consider the algorithm to be a straightforward application of the basic ideas from Time, Clocks and the Ordering of Events in a Distributed System.
- Chandy-Lamport 算法
  - Initiating a snapshot: 也就是开始创建 snapshot, 可以由系统中的任意一个进程发起
    - 进程  $P_i$  发起: 记录自己的进程状态, 同时生产一个标识信息 marker, marker 和进程通信的 message 不同
    - 将 marker 信息通过 output channel 发送给系统里面的其他进程
    - 开始记录所有 input channel 接收到的 message
  - Propagating a snapshot: 系统中其他进程开始逐个创建 snapshot 的过程
    - 对于进程  $P_j$  从 input channel  $C_{kj}$  接收到 marker 信息:
    - 如果  $P_j$  还没有记录自己的进程状态, 则
      - $P_j$  记录自己的进程状态, 同时将 channel  $C_{kj}$  置为空
      - 向 output channel 发送 marker 信息
    - 否则
      - 记录其他 channel 在收到 marker 之前的 channel 中收到所有 message
  - Terminating a snapshot: 算法结束条件
    - 所有的进程都收到 marker 信息并且记录下自己的状态和 channel 的状态 (包含的 message)

- 案例

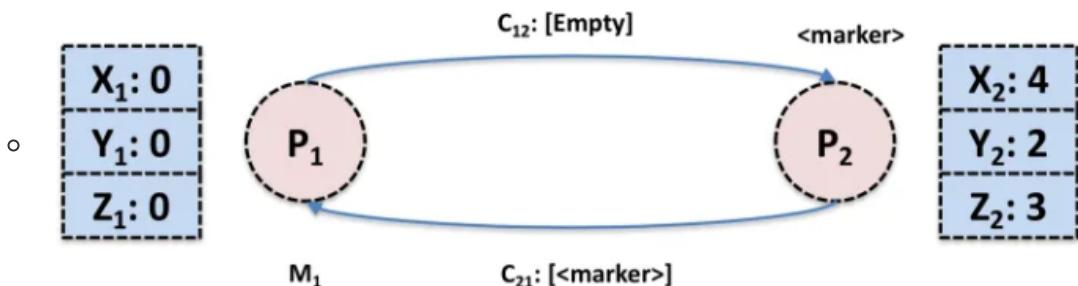
- 假设系统中包含两个进程  $P_1$  和  $P_2$ ,  $P_1$  进程状态包括三个变量  $X_1, Y_1$  和  $Z_1$ ,  $P_2$  进程包括三个变量  $X_2, Y_2$  和  $Z_2$ 。初始状态如下。



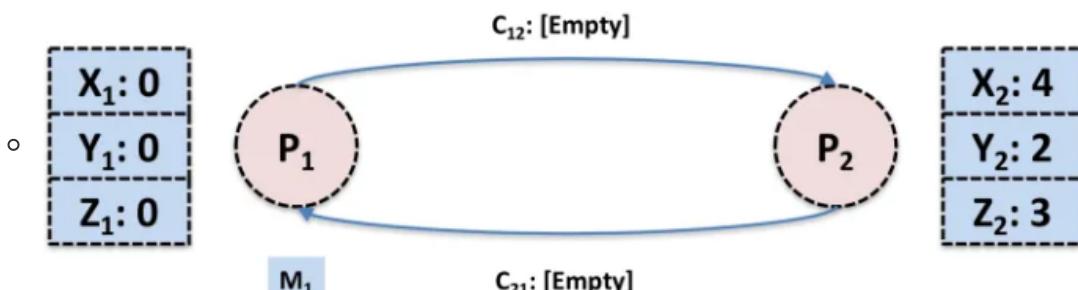
- 由  $P_1$  发起全局 Snapshot 记录,  $P_1$  先记录本身的进程状态, 然后向  $P_2$  发送 marker 信息。在 marker 信息到达  $P_2$  之前,  $P_2$  向  $P_1$  发送 message:  $M$ 。



- $P_2$  收到  $P_1$  发送过来的 marker 信息之后, 记录自己的状态。然后  $P_1$  收到  $P_2$  之前发送过来的 message:  $M$ 。对于  $P_1$  来说, 从  $P_2$  channel 发送过来的信息相当于是  $[M, \text{marker}]$ , 由于  $P_1$  已经做了 local snapshot, 所以  $P_1$  需要记录 message  $M$ 。



- 那么全局 Snapshot 就相当于下图中的蓝色部分。



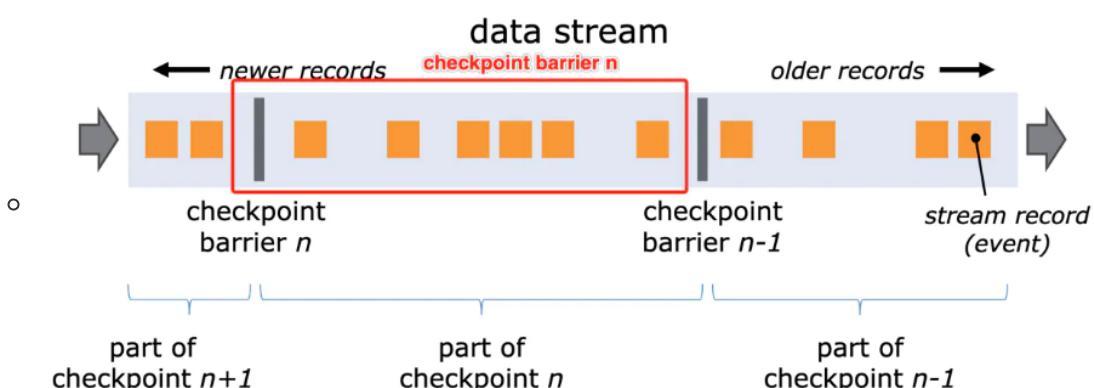
### 3.2.2. 数据屏障

#### barrier

英 [ˈbæriə(r)] ⓘ ⓘ 美 [ˈbærɪər] ⓘ ⓘ

n. 障碍; 屏障; 阻力; 关卡; 分界线; 隔阂; 难以逾越的数量  
(或水平、数目)

- 数据分割

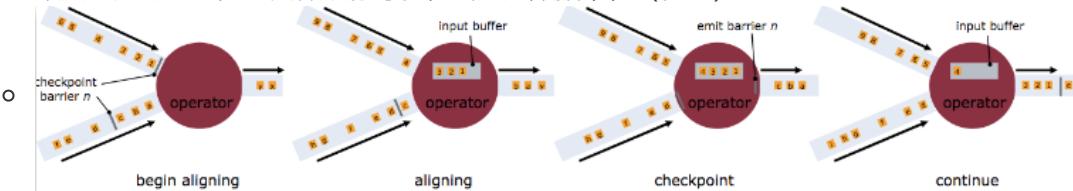


- Flink 分布式快照里面的一个核心的元素就是流屏障 (stream barrier)。这些屏障会被插入 (injected) 到数据流中，并作为数据流的一部分随着数据流动。屏障并不会持有任何数据，而是和数据一样线性的流动。可以看到屏障将数据流分成了两部分数据 (实际上是多个连续的部分)，一部分是当前快照的数据，一部分下一个快照的数据。每个屏障会带有它的快照 ID。这个快照的数据都在这个屏障的前面。
- 如果是多个输入数据流，多个数据流的屏障会被同时插入到数据流中。快照n的屏障被插入到数据流的点 (我们称之为Sn)，就是数据流中一直到的某个位置 (包含了当前时刻之前时间的所有数据)，也就是包含的这部分数据的快照。举例来说，在 Kafka 中，这个位置就是这个分区的最后一条记录的 offset。这个位置 Sn 就会上报给 checkpoint 的协调器 (Flink 的 JobManager)。
- 然后屏障开始向下流动。当一个中间的 operator 收到它的所有输入源的快照n屏障后，它就会向它所有的输出流发射一个快照n的屏障，一旦一个 sink 的 operator 收到所有输入数据流的

屏障n，它就会向checkpoint的协调器发送快照n确认。当所有的sink都确认了快照n，系统才认为当前快照的数据已经完成。

- barrier对齐

- 等到上游所有的并行子分区barrier都到齐，才去保存当前任务的状态
- 缺点：先到达的分区要做缓存等待，会造成数据堆积（背压）



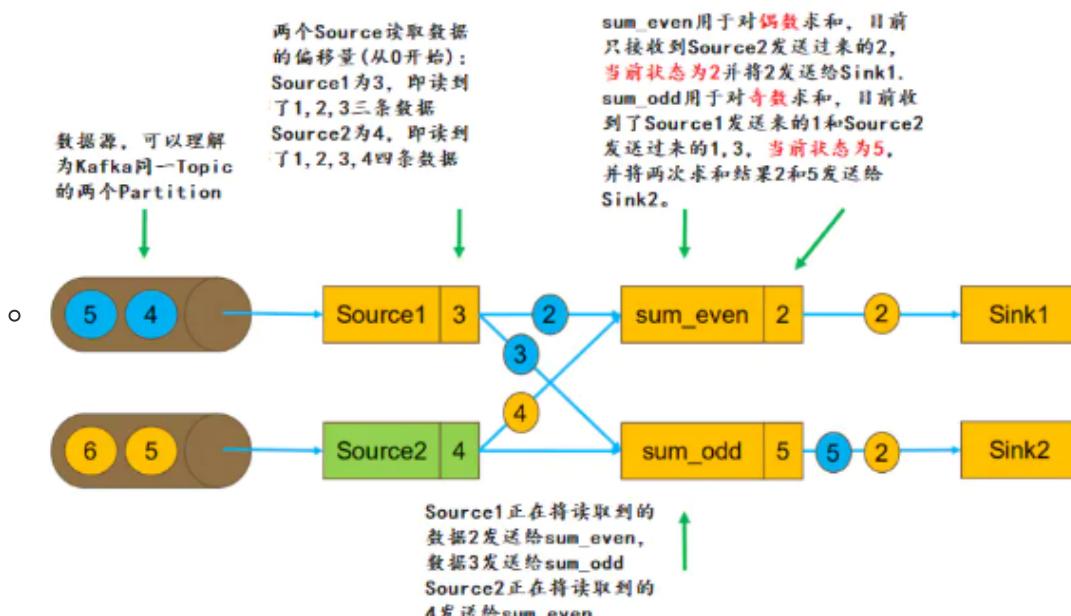
- 四大名著案例

- 《红楼梦》主要人物：贾宝玉、林黛玉、薛宝钗、贾琏、贾元春、贾迎春、贾探春、贾政、贾惜春；
- 《三国演义》主要人物：诸葛亮、曹操、关羽、孙尚香、张飞、刘备、周瑜、大乔、孙权、赵云；
- 《西游记》主要人物：唐僧、孙悟空、猪八戒、女儿国王、沙和尚、白龙马、牛夫人、牛魔王；
- 《水浒传》主要人物：宋江、卢俊义、扈三娘、吴用、武松、孙二娘、李逵、林冲、鲁智深。

### 3.2.3. 流程详解

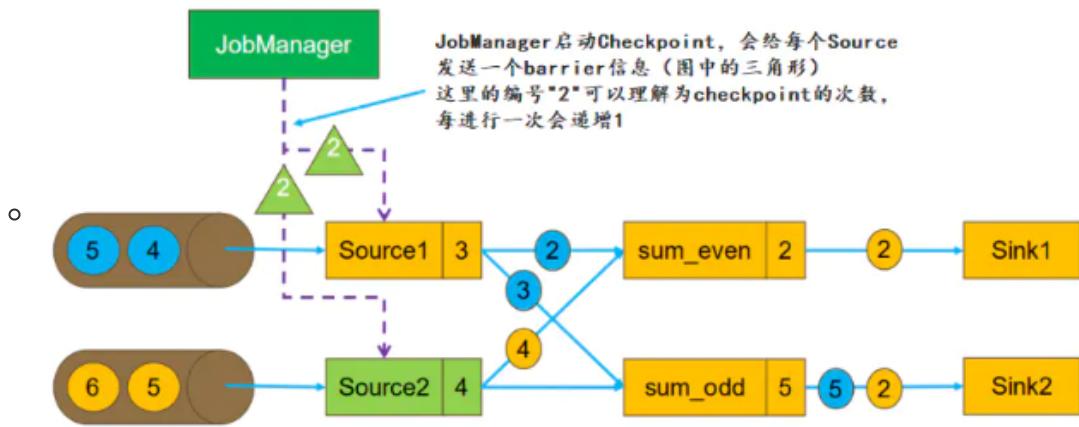
- 任务启动

- 假设任务从 Kafka 的某个 Topic 中读取数据，该Topic 有 2 个 Partition，故任务的并行度为 2。根据读取到数据的奇偶性，将数据分发到两个 task 进行求和。某一时刻，状态如下：
- Source1的偏移量为 3，即读取到了 1,2,3 三条数据。数据1已经发送到 sum\_odd。
- Source2的偏移量为 4，即读取到了 1,2,3,4 四条数据。数据1,3已经发送到sum\_odd，数据2 已经发送到sum\_even
- 此时 sum\_even 的状态为 2，sum\_odd 的状态为 5



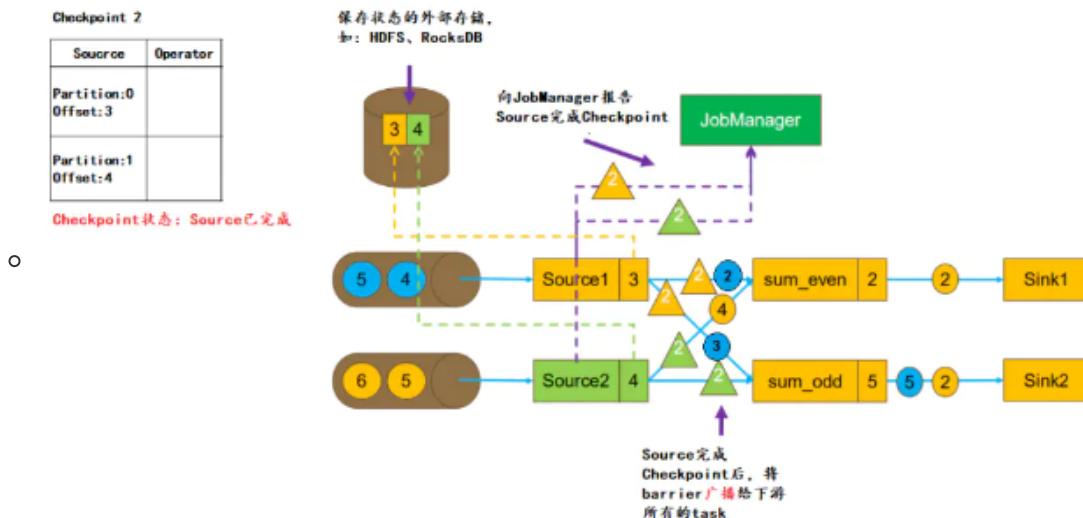
- 启动Checkpoint

- JobManager 根据 Checkpoint 间隔时间，启动 Checkpoint。此时会给每个 Source 发送一个 barrier 消息，消息中的数值表示 Checkpoint 的序号，每次启动新的 Checkpoint 该值都会递增。



- Source端

- 当Source接收到barrier消息，会将当前的状态（Partition、Offset）保存到StateBackend，然后向JobManager报告Checkpoint完成。之后Source会将barrier消息广播给下游的每一个task：

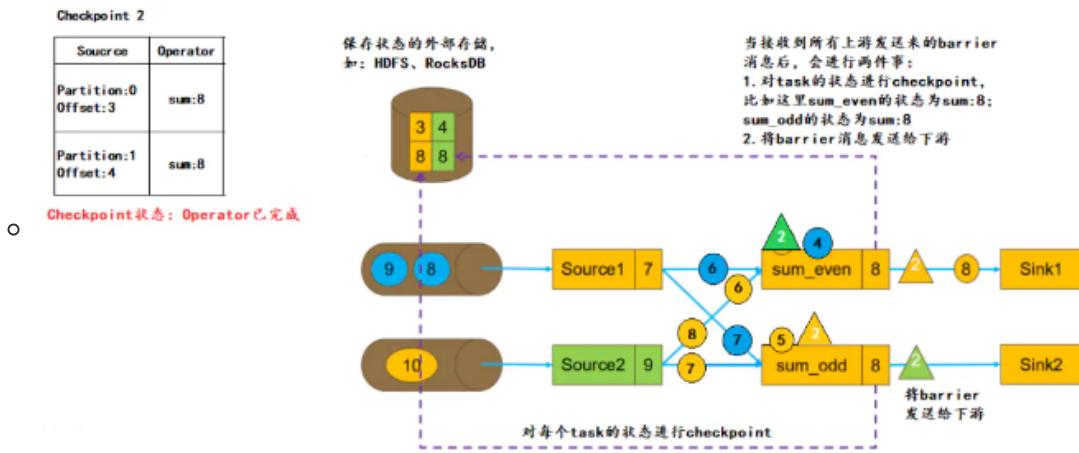


- Transformation端

- 当task接收到某个上游发送来的barrier，会将该上游barrier之前的数据继续进行处理，而barrier之后发送来的消息不会进行处理，会被缓存起来。
- barrier之前的数据属于本次Checkpoint，barrier之后的数据属于下一次Checkpoint，所以下次Checkpoint的数据是不应该在本次Checkpoint过程中被计算的，因此会将数据进行缓存。

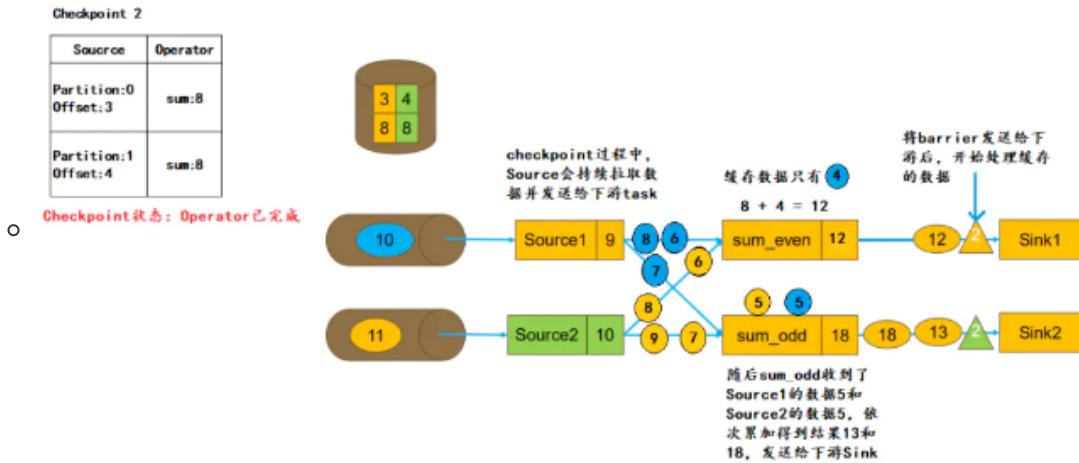
- barrier对齐

- 如果某个task有多个上游输入，如这里的 sum\_even 有两个 Source 源，当接收到其中一个 Source 的barrier后，会等待其他 Source 的 barrier 到来。在此期间，接收到 barrier 的 Source 发来的数据不会处理，只会缓存（如下图中的数据4）。而未接收到 barrier 的 Source 发来的数据依然会进行处理，直到接收到该Source 发来的 barrier，这个过程称为**barrier的对齐**。
- barrier对齐只会发生在多对一的Operator（如join）或者一对多的Operator（如repartition/shuffle）。如果是一对一的Operator，如map、flatMap 或 filter 等，则没有对齐这个概念



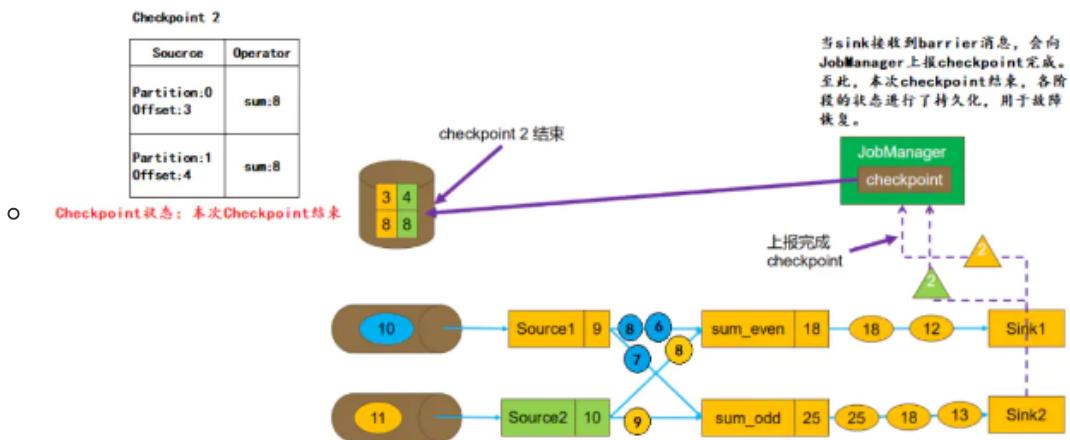
- 缓存数据

- 当task接收到所有上游发送来的barrier，即可以认为当前task收到了本次 Checkpoint 的所有数据。之后 task 会将 barrier 继续发送给下游，然后处理缓存的数据，比如这里 sum\_even 会处理 Source1 发送来的数据4. 而且，在这个过程中 Source 会继续读取数据发送给下游，并不会中断。



- Sink端

- 当sink收到barrier后，会向JobManager上报本次Checkpoint完成。至此，本次Checkpoint结束，各阶段的状态均进行了持久化，可以用于后续的故障恢复。

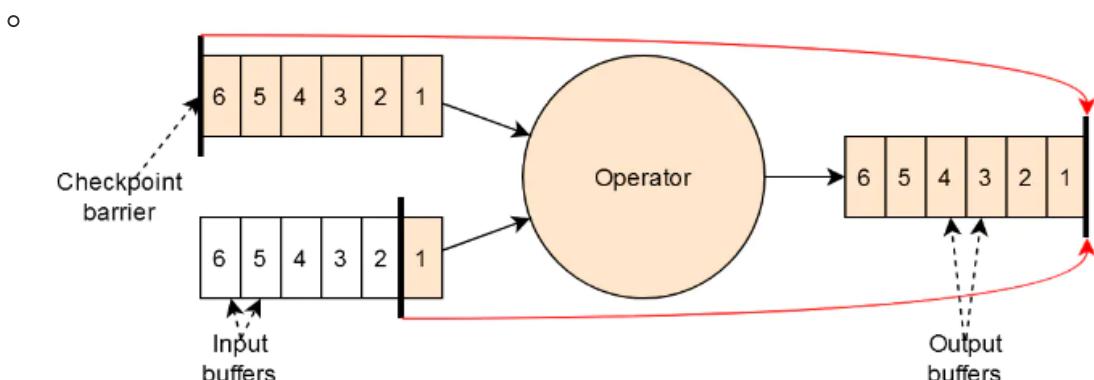


### 3.2.4. 机制革新

- 反压时无法做出 Checkpoint：

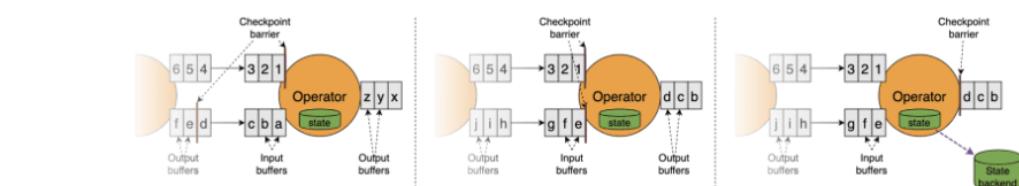
- 在反压时候 barrier 无法随着数据往下游流动，造成反压的时候无法做出 Checkpoint。但是其实在发生反压情况的时候，我们更加需要去做出对数据的Checkpoint，因为这个时候性能遇到了瓶颈，是更加容易出问题的阶段；
- Barrier 对齐阻塞数据处理：
  - 阻塞对齐对于性能上存在一定的影响；
- 恢复性能受限于 Checkpoint 间隔：
  - 在做恢复的时候，延迟受到多大的影响很多时候是取决于 Checkpoint 的间隔，间隔越大，需要 replay 的数据就会越多，从而造成中断的影响也就会越大。
  - 但是目前 Checkpoint 间隔受制于持久化操作的时间，所以没办法做的很快。
- 解决方案：Unaligned Checkpoint

- barrier 算子在到达 input buffer 最前面的时候，就会开始触发 Checkpoint 操作。它会立刻把 barrier 传到算子的 OutPut Buffer 的最前面，相当于它会立刻被下游的算子所读取到。通过这种方式可以使得 barrier 不受到数据阻塞，解决反压时候无法进行 Checkpoint 的问题。
- 当我们把 barrier 发下去后，需要做一个短暂的暂停，暂停的时候会把算子的 State 和 input output buffer 中的数据进行一个标记，以方便后续随时准备上传。对于多路情况会一直等到另外一路 barrier 到达之前数据，全部进行标注。
- 通过这种方式整个在做 Checkpoint 的时候，也不需要对 barrier 进行对齐，唯一需要做的停顿就是在整个过程中对所有 buffer 和 state 标注。这种方式可以很好的解决反压时无法做出 Checkpoint，和 Barrier 对齐阻塞数据影响性能处理的问题。
- 

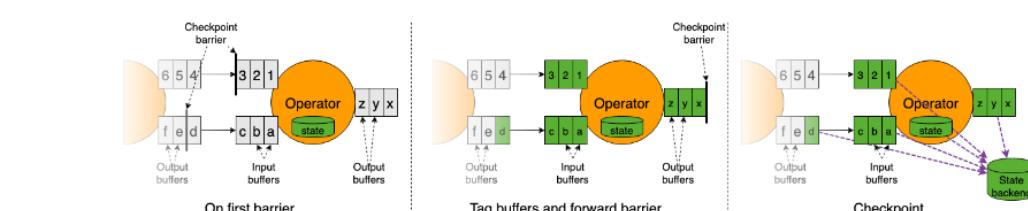


- 差异
  - CheckPoint的触发是在接收到第一个 Barrier[对不齐] 时还是在接收到最后一个 Barrier [对齐]时。
  - 是否需要阻塞已经接收到 Barrier 的 Channel 的计算。

Alignment checkpoint



Unalignment checkpoint



### 3.2.5. SavePoint

- Savepoint 作为实时任务的全局镜像，其在底层使用的代码和Checkpoint的代码是一样的
- Savepoint 是依据 Flink checkpointing 机制所创建的流作业执行状态的一致镜像；
  - Checkpoint 的主要目的是为意外失败的作业提供恢复机制(如 tm/jm 进程挂了)。
  - Checkpoint 的生命周期由 Flink 管理，即 Flink 创建，管理和删除 Checkpoint - 无需用户交互。
  - Savepoint 由用户创建，拥有和删除。他们的用例是计划的，手动备份和恢复。
  - Savepoint 应用场景，升级 Flink 版本，调整用户逻辑，改变并行度，以及进行红蓝部署等。Savepoint 更多地关注可移植性
- Savepoint触发方式触发方式目前有三种
  - 使用 **flink savepoint** 命令触发 Savepoint,其是在程序运行期间触发 savepoint。
  - 使用 **flink cancel -s** 命令，取消作业时，并触发 Savepoint。
  - 使用 Rest API 触发 Savepoint，格式为：**\*/jobs/:jobid /savepoints\***
- Savepoint注意点
  - 由于 Savepoint 是程序的全局状态，对于某些状态很大的实时任务，当我们触发 Savepoint，可能会对运行着的实时任务产生影响，个人建议如果对于状态过大的实时任务，触发 Savepoint 的时间，不要太过频繁。根据状态的大小，适当的设置触发时间。
  - 当我们从 Savepoint 进行恢复时，需要检查这次 Savepoint 目录文件是否可用。可能存在你上次触发 Savepoint 没有成功，导致 HDFS 目录上面 Savepoint 文件不可用或者缺少数据文件等，这种情况下，如果在指定损坏的 Savepoint 的状态目录进行状态恢复，任务会启动不起来。

## 3.3. 容错策略

- 当 Task 发生故障时，Flink 需要重启出错的 Task 以及其他受到影响的 Task，以使得作业恢复到正常执行状态。
- Flink 通过重启策略和故障恢复策略来控制 Task 重启：
  - 重启策略决定是否可以重启以及重启的间隔；
  - 故障恢复策略决定哪些 Task 需要重启。

### 3.3.1. 重启策略

- Flink 作业如果没有定义重启策略，则会遵循集群启动时加载的默认重启策略。如果提交作业时设置了重启策略，该策略将覆盖掉集群的默认策略。
- 通过 Flink 的配置文件 `flink-conf.yaml` 来设置默认的重启策略。配置参数 `restart-strategy` 定义了采取何种策略。
  - 如果没有启用 checkpoint，就采用“不重启”策略。
  - 如果启用了 checkpoint 且没有配置重启策略，那么就采用固定延时重启策略，此时最大尝试重启次数由 `Integer.MAX_VALUE` 参数设置。
- 重启策略
-

Key	Default	Type	Description
restart-strategy	(none)	String	<p>Defines the restart strategy to use in case of job failures. Accepted values are:</p> <ul style="list-style-type: none"> <li>• <code>none</code>, <code>off</code>, <code>disable</code>: No restart strategy.</li> <li>• <code>fixeddelay</code>, <code>fixed-delay</code>: Fixed delay restart strategy. More details can be found <a href="#">here</a>.</li> <li>• <code>failurerate</code>, <code>failure-rate</code>: Failure rate restart strategy. More details can be found <a href="#">here</a>.</li> <li>• <code>exponentialdelay</code>, <code>exponential-delay</code>: Exponential delay restart strategy. More details can be found <a href="#">here</a>.</li> </ul> <p>If checkpointing is disabled, the default value is <code>none</code>. If checkpointing is enabled, the default value is <code>fixed-delay</code> with <code>Integer.MAX_VALUE</code> restart attempts and '1' s' delay.</p>

- 固定延迟重启策略
  - Fixed Delay Restart Strategy
  - 固定延时重启策略按照给定的次数尝试重启作业。如果尝试超过了给定的最大次数，作业将最终失败。
  - 在连续的两次重启尝试之间，重启策略等待一段固定长度的时间。
  - ```
//配置文件 restart-strategy: fixed-delay
ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
env.setRestartStrategy(RestartStrategies.fixedDelayRestart(
    3, // 尝试重启的次数
    Time.of(10, TimeUnit.SECONDS) // 延时
));
```

| o | Key                                   | Default | Type     | Description                                                                                                                                                                                                                                                                                                                                                                        |
|---|---------------------------------------|---------|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ◦ | restart-strategy.fixed-delay.attempts | 1       | Integer  | The number of times that Flink retries the execution before the job is declared as failed if <code>restart-strategy</code> has been set to <code>fixed-delay</code> .                                                                                                                                                                                                              |
| ◦ | restart-strategy.fixed-delay.delay    | 1 s     | Duration | Delay between two consecutive restart attempts if <code>restart-strategy</code> has been set to <code>fixed-delay</code> . Delaying the retries can be helpful when the program interacts with external systems where for example connections or pending transactions should reach a timeout before re-execution is attempted. It can be specified using notation: "1 min", "20 s" |

- 故障率重启策略
  - Failure Rate Restart Strategy
  - 故障率重启策略在故障发生之后重启作业，但是当故障率（每个时间间隔发生故障的次数）超过设定的限制时，作业会最终失败。
  - 在连续的两次重启尝试之间，重启策略等待一段固定长度的时间。

- //配置文件restart-strategy: failure-rate

```
ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
env.setRestartStrategy(RestartStrategies.failureRateRestart(
    3, // 每个时间间隔的最大故障次数
    Time.of(5, TimeUnit.MINUTES), // 测量故障率的时间间隔
    Time.of(10, TimeUnit.SECONDS) // 延时
));
```

| Key                                                     | Default | Type     | Description                                                                                                                                                                     |
|---------------------------------------------------------|---------|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| restart-strategy.failure-rate.delay                     | 1 s     | Duration | Delay between two consecutive restart attempts if <code>restart-strategy</code> has been set to <code>failure-rate</code> . It can be specified using notation: "1 min", "20 s" |
| restart-strategy.failure-rate.failure-rate-interval     | 1 min   | Duration | Time interval for measuring failure rate if <code>restart-strategy</code> has been set to <code>failure-rate</code> . It can be specified using notation: "1 min", "20 s"       |
| restart-strategy.failure-rate.max-failures-per-interval | 1       | Integer  | Maximum number of restarts in given time interval before failing a job if <code>restart-strategy</code> has been set to <code>failure-rate</code> .                             |

- 不重启策略
  - No Restart Strategy
  - 作业直接失败，不尝试重启。
  - //配置文件restart-strategy: none

```
ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
env.setRestartStrategy(RestartStrategies.noRestart());
```
- 备用重启策略
  - Fallback Restart Strategy
  - 使用群集定义的重启策略。这对于启用了 checkpoint 的流处理程序很有帮助。如果没有定义其他重启策略，默认选择固定延时重启策略。

### 3.3.2. 恢复策略

- Flink 支持多种不同的故障恢复策略，该策略需要通过 Flink 配置文件 `flink-conf.yaml` 中的 `jobmanager.execution.failover-strategy` 配置项进行配置

| 故障恢复策略          | jobmanager.execution.failover-strategy 配置值 |
|-----------------|--------------------------------------------|
| 全图重启            | full                                       |
| 基于 Region 的局部重启 | region                                     |

- 全部重启
  - Restart All Failover Strategy
  - 在全图重启故障恢复策略下，Task 发生故障时会重启作业中的所有 Task 进行故障恢复。
- 当前Region
  - Restart Pipelined Region Failover Strategy
  - 该策略会将作业中的所有 Task 划分为数个 Region。当有 Task 发生故障时，它会尝试找出进行故障恢复需要重启的最小 Region 集合。
  - 相比于全局重启故障恢复策略，这种策略在一些场景下的故障恢复需要重启的 Task 会更少。
  - 此处 Region 指以 Pipelined 形式进行数据交换的 Task 集合。
  - 重启的 Region 的判断逻辑
    - 出错 Task 所在 Region 需要重启。
    - 如果要重启的 Region 需要消费的数据有部分无法访问（丢失或损坏），产出该部分数据的 Region 也需要重启。
    - 需要重启的 Region 的下游 Region 也需要重启。这是出于保障数据一致性的考虑，因为一些非确定性的计算或者分发会导致同一个 Result Partition 每次产生时包含的数据都不相同。

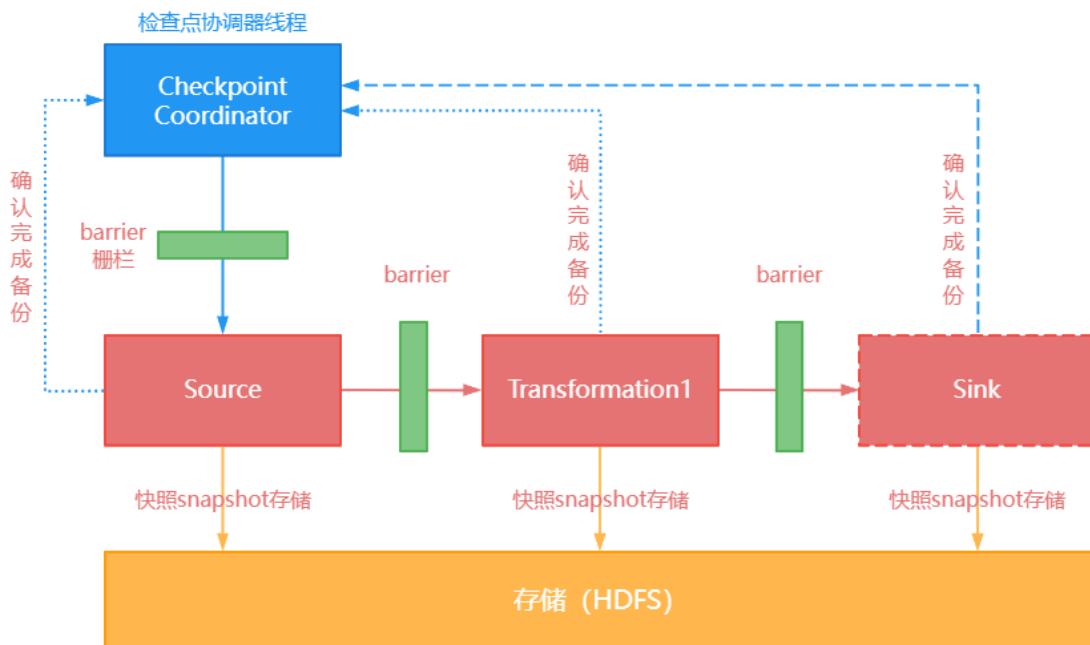
### 3.4. Flink Source

- Flink 的很多 source 算子都能为 EOS 提供保障，如 kafka Source：
  - 能够记录偏移量
  - 能够重放数据
  - 将偏移量记录在 state 中，与下游的其他算子的 state 一起，经由 checkpoint 机制实现了“状态数据的”快照统一
- 精确一次&有效一次
  - 有些人可能认为『精确一次』描述了事件处理的保证，其中流中的每个事件只被处理一次。实际上，没有引擎能够保证正好只处理一次。在面对任意故障时，不可能保证每个算子中的用户定义逻辑在每个事件中只执行一次，因为用户代码被部分执行的可能性是永远存在的。
  - 那么当引擎声明『精确一次』处理语义时，它们能保证什么呢？如果不能保证用户逻辑只执行一次，那么什么逻辑只执行一次？当引擎声明『精确一次』处理语义时，它们实际上是在说，它们可以保证引擎管理的状态更新只提交一次到持久的后端存储。
  - 事件的处理可以发生多次，但是该处理的效果只在持久后端状态存储中反映一次。因此，我们认为有效地描述这些处理语义最好的术语是『有效一次』（effectively once）。
- 有效一次的实现策略
  - At-least-once + 去重
    - 每个算子维护一个事务日志，跟踪已处理的事件
    - 重放失败事件，在时间进入下一个算子之前，移除重复事件
  - At-least-once + 幂等
    - 依赖Sink端存储的去重性和数据特性
  - 分布式快照
    - 借助于Flink 本身自带的Checkpoint

| Exactly Once 实现方式  | 优点                                                                                          | 缺点                                                                                                                                                                  |
|--------------------|---------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| At least once + 去重 | <ul style="list-style-type: none"> <li>故障对性能的影响是局部的</li> <li>故障的影响不一定会随着拓扑的大小而增加</li> </ul> | <ul style="list-style-type: none"> <li>可能需要大量的存储和基础设施来支持</li> <li>每个算子的每个事件的性能开销</li> </ul>                                                                         |
| At least once + 幂等 | <ul style="list-style-type: none"> <li>实现简单，开销较低</li> </ul>                                 | <ul style="list-style-type: none"> <li>依赖存储特性和数据特征</li> </ul>                                                                                                       |
| 分布式快照              | <ul style="list-style-type: none"> <li>较小的性能和资源开销</li> </ul>                                | <ul style="list-style-type: none"> <li>barrier 同步</li> <li>任何算子发生故障，都需要发生全局暂停和状态回滚（Region Failover: <a href="#">FLINK-4256</a>）</li> <li>拓扑越大，对性能的潜在影响越大</li> </ul> |

### 3.5. Flink Operator

- 算子状态的 EOS 语义保证
  - 基于分布式快照算法: (Chandy-Lamport) , flink 实现了整个数据流中各算子的状态数据快照统一；
  - 既: 一次 checkpoint 后所持久化的各算子的状态数据，确保是经过了相同数据的影响；
  - 如果这条（批）数据在中间任何过程失败，则重启恢复后，所有算子的 state 数据都能回到这条数据从未处理过时的状态
- 同 Spark 相比，Spark 仅仅是针对 Driver 的故障恢复 Checkpoint。而 Flink 的快照可以到算子级别，并且对全局数据也可以做快照。
- 检查点协调器线程



### 3.6. Flink Sink

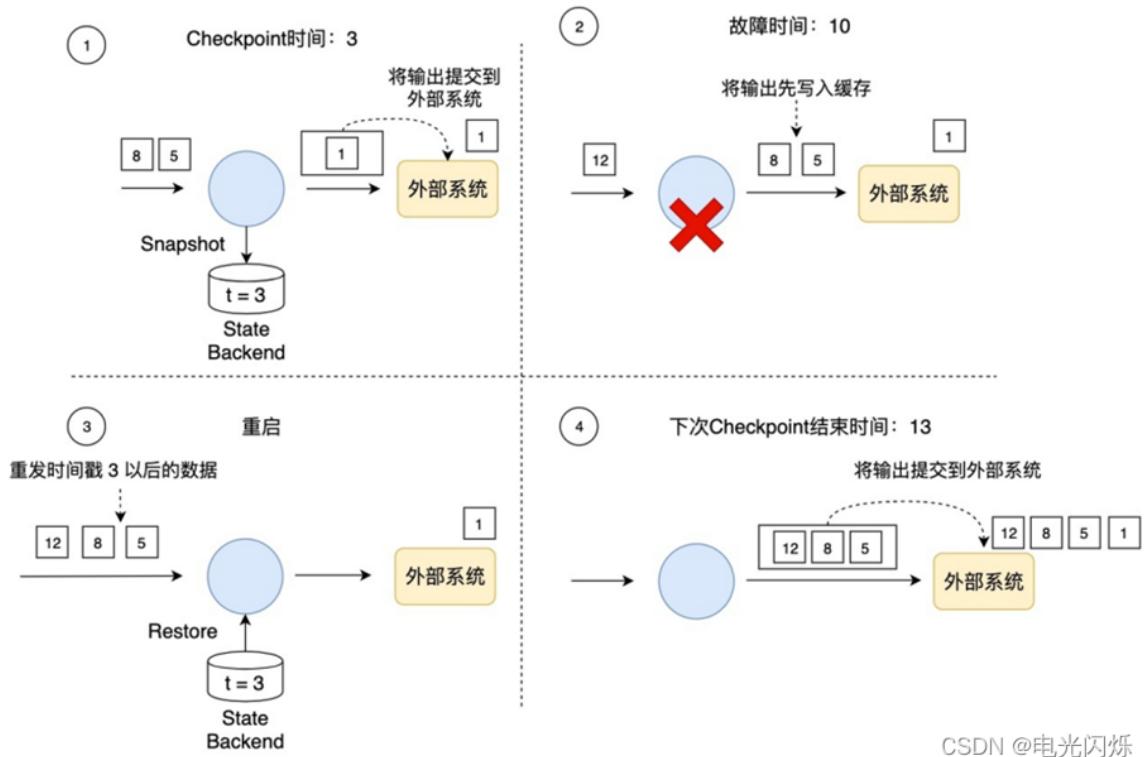
- Sink 端主要的问题是，作业失败重启时，数据重放可能造成最终目标存储中被写入了重复数据；
- Flink 中也设计了相应机制来确保 EOS
  - 采用幂等写入方式
  - 采用事务写入方式
    - 采用预写日志提交方式
  - 2PC 提交方式

### 3.6.1. 幂等写入

- 幂等写入 (Idempotent Writes)
- 幂等写操作是指：任意多次向一个系统写入数据，只对目标系统产生一次结果影响。
- 例如，重复向一个HashMap里插入同一个Key-Value二元对，第一次插入时这个HashMap发生变化，后续的插入操作不会改变HashMap的结果，这就是一个幂等写操作。HBase、Redis和Cassandra这样的KV数据库一般经常用来作为Sink，用以实现端到端的Exactly-Once。
- 注意点1，并不是说一个KV数据库就百分百支持幂等写。幂等写对KV对有要求，那就是Key-Value必须是可确定性 (Deterministic) 计算的。假如我们设计的Key是：name + currentTimestamp，每次执行数据重发时，生成的Key都不相同，会产生多次结果，整个操作不是幂等的。因此，为了追求端到端的Exactly-Once，我们设计业务逻辑时要尽量使用确定性的计算逻辑和数据模型。
- 注意点2，对于幂等写入，遇到故障进行恢复时，有可能会出现短暂的不一致。因为保存点完成之后到发生故障之间的数据，其实已经写入了一遍，回滚的时候并不能消除它们。如果有一个外部应用读取写入的数据，可能会看到奇怪的现象：短时间内，结果会突然“跳回”到之前的某个值，然后“重播”一段之前的数据。不过当数据的重放逐渐超过发生故障的点的时候，最终的结果还是一致的。

### 3.6.2. 事务写入之预写日志

- 事务写入 (Transactional Writes)
  - 事务 (transaction) 是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所做的所有更改都会被撤消。
  - 事务有四个基本特性：原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)和持久性(Durability)，这就是著名的 ACID。
- Flink借鉴了数据库中的事务处理技术，同时结合自身的Checkpoint机制来保证Sink只对外部输出产生一次影响。大致的流程如下：
  - Flink先将待输出的数据保存下来暂时不向外部系统提交，等到Checkpoint结束时，Flink上下游所有算子的数据都是一致的时候，Flink将之前保存的数据全部提交并commit到外部系统。换句话说，只有经过Checkpoint确认的数据才向外部系统写入。
  - 如下图所示，如果使用事务写，那只把时间戳3之前的输出提交到外部系统，时间戳3以后的数据（例如时间戳5和8生成的数据）暂时保存下来，等待下次Checkpoint时一起写入到外部系统。这就避免了时间戳5这个数据产生多次结果，多次写入到外部系统。

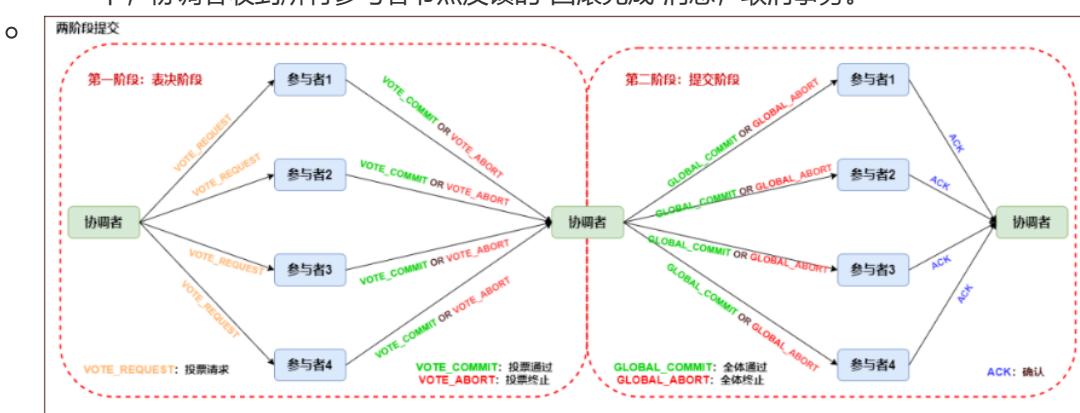


CSDN @电光闪烁

- 在事务写的具体实现上，Flink目前提供了两种方式：
  - 预写日志 (Write-Ahead-Log, WAL)
  - 两阶段提交 (Two-Phase-Commit, 2PC)
- 预写日志
  - 事务提交是需要外部存储系统支持事务的，否则没有办法真正实现写入的回撤。那对于一般不支持事务的存储系统，预写日志 (WAL) 就是一种非常简单的方式。
    - 先把结果数据作为日志 (log) 状态保存起来
    - 进行检查点保存时，也会将这些结果数据一并做持久化存储
    - 在收到检查点完成的通知时，将所有结果一次性写入外部系统。
  - 需要注意的是，预写日志这种一批写入的方式，有可能会写入失败；所以在执行写入动作之后，必须等待发送成功的返回确认消息。在成功写入所有数据后，在内部再次确认相应的检查点，这才代表着检查点的真正完成。这里需要将确认信息也进行持久化保存，在故障恢复时，只有存在对应的确认信息，才能保证这批数据已经写入，可以恢复到对应的检查点位置。
  - 但这种“再次确认”的方式，也会有一些缺陷。如果我们的检查点已经成功保存、数据也成功地一批写入到了外部系统，但是最终保存确认信息时出现了故障，Flink 最终还是会认为没有成功写入。于是发生故障时，不会使用这个检查点，而是需要回退到上一个；这样就会导致这批数据的重复写入。
- 这两种方式区别主要在于：
  - WAL方式通用性更强，适合几乎所有外部系统，但也不能提供百分百端到端的Exactly-Once，因为WAL预写日志会先写内存，而内存是易失介质。
  - 如果外部系统自身就支持事务（比如MySQL、Kafka），可以使用2PC方式，可以提供百分百端到端的Exactly-Once。
- 事务写的方式能提供端到端的Exactly-Once一致性，它的代价也是非常明显的，就是牺牲了延迟。输出数据不再是实时写入到外部系统，而是分批次地提交。目前来说，没有完美的故障恢复和Exactly-Once保障机制，对于开发者来说，需要在不同需求之间权衡。

### 3.6.3. 事务写入之两阶段提交

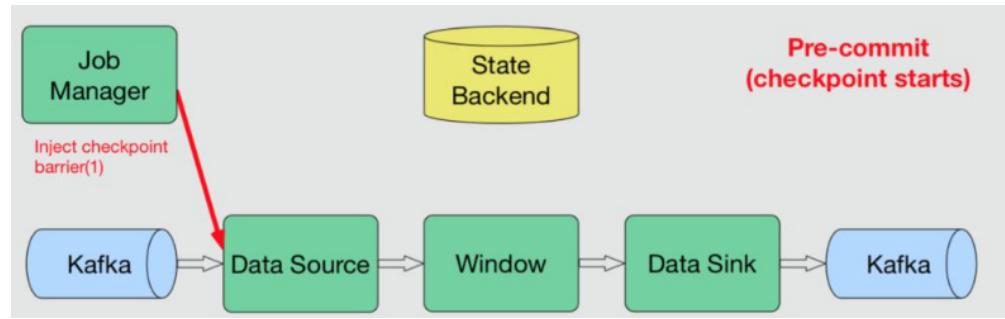
- 两阶段提交(Two-phase Commit, 简称2PC)
- Flink社区将两阶段提交协议中的公共逻辑进行提取和封装，发布了可供用户自定义实现特定方法来达到flink EOS特点的TwoPhaseCommitSinkFunction
- 两阶段提交可以归纳为一目的两角色三条件
  - 一目的：
    - 分布式系统架构下的所有节点在进行事务提交时要保持一致性（即要么全部成功，要么全部失败）
  - 两角色：
    - 协调者 (Coordinator) , 负责统筹并下达命令工作
    - 参与者 (Participants) , 负责认真干活并响应协调者的命令。
  - 三条件：
    2. 分布式系统中必须存在一个协调者节点和多个参与者节点，且所有节点之间可以相互正常通信；
    3. 所有节点都采用预写日志方式，且日志可以可靠存储。
    4. 所有节点不会永久性损坏，允许可恢复性的短暂损坏。
- 两阶段提交，顾名思义，即分两个阶段commit：preCommit和Commit。
  - preCommit阶段
    - 协调者向所有参与者发起请求，询问是否可以执行提交操作，并开始等待所有参与者的响应。
    - 所有参与者节点执行协调者询问发起为止的所有事务操作，并将undo和redo信息写入日志进行持久化。
    - 所有参与者响应协调者发起的询问。对于每个参与者节点，如果他的事务操作执行成功，则返回“同意”消息；反之，返回“终止”消息。
  - commit阶段
    - 如果协调者获取到的所有参与者节点返回的消息都为“同意”时，协调者向所有参与者节点发送“正式提交”的请求（成功情况）；反之，如果任意一个参与者节点预提交阶段返回的响应消息为“终止”，或者协调者询问阶段超时，导致没有收到所有的参与者节点的响应，那么，协调者向所有参与者节点发送“回滚提交”的请求（失败情况）。
    - 成功情况下，所有参与者节点正式完成操作，并释放放在整个事务期间占用的资源；反之，失败情况下，所有参与者节点利用之前持久化的预写日志进行事务回滚操作，并释放放在整个事务期间占用的资源。
    - 成功情况下，所有参与者节点向协调者节点发送“事务完成”消息；失败情况下，所有参与者节点向协调者节点发送“回滚完成”消息。
    - 成功情况下，协调者收到所有参与者节点反馈的“事务完成”消息，完成事务；失败情况下，协调者收到所有参与者节点反馈的“回滚完成”消息，取消事务。



- 详细流程
  - Flink的两阶段提交：从 Flink 程序启动到消费 Kafka 数据，最后到 Flink 将数据 Sink 到 Kafka 为止，来分析 Flink 的精准一次处理

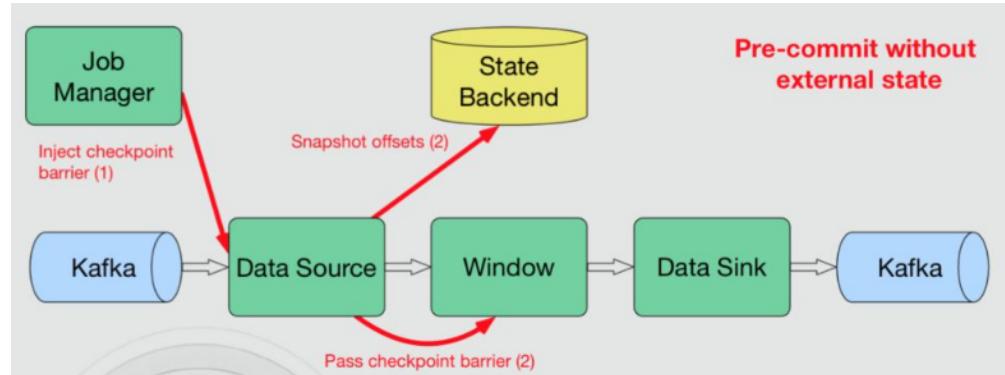
- 当 Checkpoint 启动时

- JobManager 会将检查点分界线 (checkpoint barrier) 注入数据流, checkpoint barrier 会在算子间传递下去



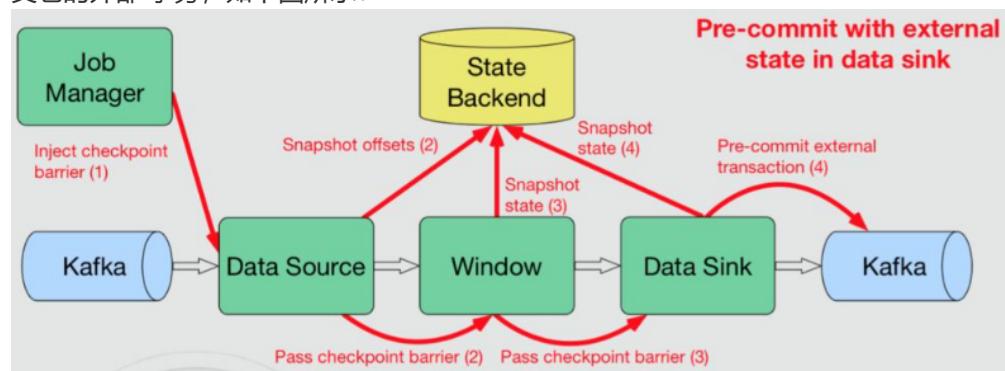
- Source 端

- Flink Kafka Source 负责保存 Kafka 消费 offset, 当 Checkpoint 成功时 Flink 负责提交这些写入, 否则就终止取消掉它们, 当 Checkpoint 完成位移保存, 它会将 checkpoint barrier (检查点分界线) 传给下一个 Operator, 然后每个算子会对当前的状态做个快照, 保存到状态后端 (State Backend)。
- 对于 Source 任务而言, 就会把当前的 offset 作为状态保存起来。下次从 Checkpoint 恢复时, Source 任务可以重新提交偏移量, 从上次保存的位置开始重新消费数据, 如下图所示:



- Sink 端:

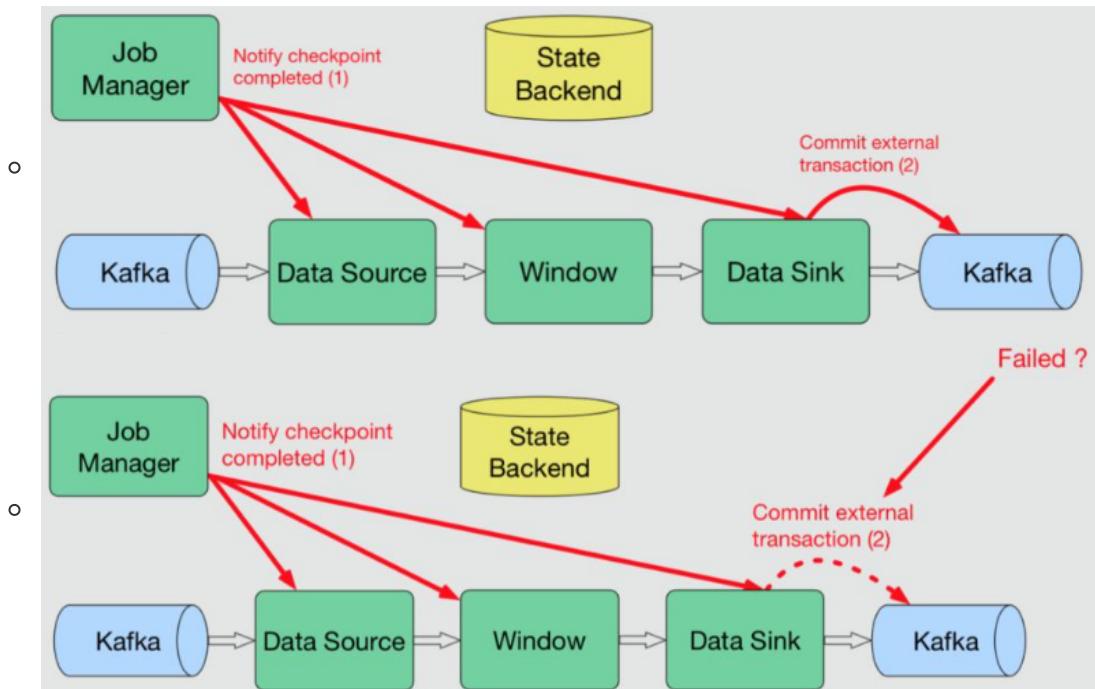
- 从 Source 端开始, 每个内部的 transform 任务遇到 checkpoint barrier 时, 都会把状态存到 Checkpoint 里。
- 数据处理完毕到 Sink 端时, Sink 任务首先把数据写入外部 Kafka, 这些数据都属于预提交的事务 (还不能被消费)
- 此时的 Pre-commit 预提交阶段下 Data Sink 在保存状态到状态后端的同时还必须预提交它的外部事务, 如下图所示:



- 任务完成时:

- 当所有算子任务的快照完成 (所有创建的快照都被视为是 Checkpoint 的一部分), 也就是这次的 Checkpoint 完成时, JobManager 会向所有任务发通知, 确认这次 Checkpoint 完成, 此时 Pre-commit 预提交阶段才算完成。才正式到两阶段提交协议的第二个阶段: commit 阶段。该阶段中 JobManager 会为应用中每个 Operator 发起 Checkpoint 已完成的回调逻辑。

- 本例中的 Data Source 和窗口操作无外部状态，因此在该阶段，这两个 Operator 无需执行任何逻辑，但是 Data Sink 是有外部状态的，此时我们必须提交外部事务，当 Sink 任务收到确认通知，就会正式提交之前的事务，Kafka 中未确认的数据就改为“已确认”，数据就真正可以被消费了，如下图所示：



- 总结

- 注：Flink 由 JobManager 协调各个 TaskManager 进行 Checkpoint 存储，Checkpoint 保存在 StateBackend（状态后端）中，默认 StateBackend 是内存级的，也可以改为文件级的进行持久化保存
- Flink 消费到 Kafka 数据之后，就会开启一个 Kafka 的事务，正常写入 Kafka 分区日志标记但未提交，也就是预提交（Per-commit）
- 一旦所有的 Operator 完成各自的 Per-commit，他们会发起一个 commit 操作
- 如果有任意一个 Per-commit 失败，所有其他的 Per-commit 必须停止，并且 Flink 会回滚到最近成功完成的 CheckPoint
- 当所有的 Operator 完成任务时，Sink 端就收到 checkpoint barrier（检查点分界线），Sink 保存当前状态，存入 Checkpoint，通知 JobManager，并提交外部事物，用于提交外部检查点的数据
- JobManager 收到所有任务通知，发出确认信息，表示 Checkpoint 已经完成，Sink 收到 JobManager 的确认信息，正式提交这段时间的数据
- 外部系统（Kafka）关闭事务，提交的数据可以正常消费了

## 3.7. Kafka之End-to-End

### 3.7.1. 版本说明

- Flink 1.4 版本之前，支持 Exactly Once 语义，仅限于应用内部。
- Flink 1.4 版本之后，通过两阶段提交（TwoPhaseCommitSinkFunction）支持 End-To-End Exactly Once。
- Kafka 要求 0.11+，Kafka 在最近的 0.11 版本中添加了对事务的支持。

### 3.7.2. 实现逻辑

- 在 Flink 中的 Two-Phase-Commit-2PC 两阶段提交的实现方法被封装到了 TwoPhaseCommitSinkFunction 这个抽象类中

- // This is a recommended base class for all of the sinkFunction that intend to implement exactly-once semantic.

```

//methods that should be implemented in child class to support two
phase commit
org.apache.flink.streaming.api.functions.sink.TwoPhaseCommitsinkFunction

//write value within a transaction.
protected abstract void invoke(TXN transaction, IN value, Context
context) throws Exception;

//Method that starts a new transaction.
//Returns: newly created transaction.
//在开启事务之前，我们在目标文件系统的临时目录中创建一个临时文件，后面在处理数据时将数
据写入此文件;
protected abstract TXN beginTransaction() throws Exception;

//Pre commit previously created transaction. Pre commit must make all
of the necessary steps to prepare the transaction for a commit that
might happen in the future. After this point the transaction might
still be aborted, but underlying implementation must ensure that commit
calls on already pre committed transactions will always succeed.
//在预提交阶段，刷写(flush)文件，然后关闭文件，之后就不能写入到文件了，我们还将为属
于下一个检查点的任何后续写入启动新事务;
protected abstract void preCommit(TXN transaction) throws Exception;

//Commit a pre-committed transaction. If this method fail, Flink
application will be restarted and recoverAndCommit(Object) will be
called again for the same transaction.
//在提交阶段，我们将预提交的文件原子性移动到真正的目标目录中，请注意，这会增加输出数据
可见性的延迟;
protected abstract void commit(TXN transaction);

//Abort a transaction.
//在中止阶段，我们删除临时文件。
protected abstract void abort(TXN transaction);

```

### 3.7.3. 代码实现

- Kafka-->Flink-->Kafka

```

• import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.RichMapFunction;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.time.Time;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.connector.base.DeliveryGuarantee;
import org.apache.flink.connector.kafka.sink.KafkaRecordSerializationSchema;
import org.apache.flink.connector.kafka.sink.KafkaSink;
import org.apache.flink.connector.kafka.source.KafkaSource;
import
org.apache.flink.connector.kafka.source.enumerator.initializer.OffsetsInitia
lizer;
import org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackend;

```

```
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.concurrent.TimeUnit;

/**
 * @Description : Source Kafka -->Flink -->Sink Kafka
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class HelloEosUseKFK {
    public static void main(String[] args) throws Exception {
        //运行环境并设置CheckPoint【开启】【最小间隔】【错误容忍】【精确一次】【超时时间】【并行度】【重启策略】
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(2);
        environment.enableCheckpointing(5000);

        environment.getCheckpointConfig().setMinPauseBetweenCheckpoints(1000);

        environment.getCheckpointConfig().setTolerableCheckpointFailureNumber(0);

        environment.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EX
ACTLY_ONCE);
        environment.getCheckpointConfig().setCheckpointTimeout(30000);
        environment.getCheckpointConfig().setMaxConcurrentCheckpoints(1);

        environment.setRestartStrategy(RestartStrategies.fixedDelayRestart(3,
Time.of(10, TimeUnit.SECONDS)));
        //本地状态维护
        environment.setStateBackend(new EmbeddedRocksDBStateBackend());
        //远程状态备份

        environment.getCheckpointConfig().setCheckpointStorage("hdfs://node02:8020/
flink/checkpoints");

        //KafkaSource 数据格式[word]普通字符串
        KafkaSource<String> kafkaSourceSetting = KafkaSource.
<String>builder()
        .setBootstrapServers("node01:9092,node02:9092,node03:9092")
        .setTopics("t_kafka_source")
        .setGroupId("liyidadi")
        .setStartingOffsets(OffsetsInitializer.latest())
        .setValueOnlyDeserializer(new SimpleStringSchema())
        .setProperty("partition.discovery.interval.ms", "10000")
        .setProperty("commit.offsets.on.checkpoint", "true")
        .build();
        DataStreamSource<String> kafkaSource =
environment.fromSource(kafkaSourceSetting, WatermarkStrategy.noWatermarks(),
"Kafka Source");

        //Flink Transformation
```

```

        singleOutputStreamOperator<String> transformation =
kafkaSource.keyBy(word -> word)
        .map(new RichMapFunction<String, String>() {
            private ValueState<Integer> valueState;
            private int countBuffer;

            @Override
            public String map(String value) throws Exception {
                //累加器叠加
                this.countBuffer++;
                //更新状态
                this.valueState.update(countBuffer);
                //返回结果
                return "value:" + this.countBuffer;
            }

            @Override
            public void open(Configuration parameters) throws
Exception {
                //初始化
                ValueStateDescriptor<Integer> valueStateDescriptor =
new ValueStateDescriptor<Integer>("countBuffer", Types.INT);
                this.valueState =
getRuntimeContext().getState(valueStateDescriptor);
                //恢复默认值
                this.countBuffer = this.valueState.value();
                System.out.println("HelloEosUseKFK.open[" +
this.valueState + "][" + this.valueState + "]");
            }
        });

        //Kafkasink
        Kafkasink<String> kafkasinkSetting = KafkaSink.<String>builder()
            .setBootstrapServers("node01:9092,node02:9092,node03:9092")

            .setRecordSerializer(KafkaRecordSerializationSchema.builder()
                .setTopic("t_kafka_sink")
                .setValueSerializationSchema(new
SimpleStringSchema())
                .build()
            )
            .setDeliveryGuarantee(DeliveryGuarantee.EXACTLY_ONCE)
            .build();
        transformation.sinkTo(kafkasinkSetting);

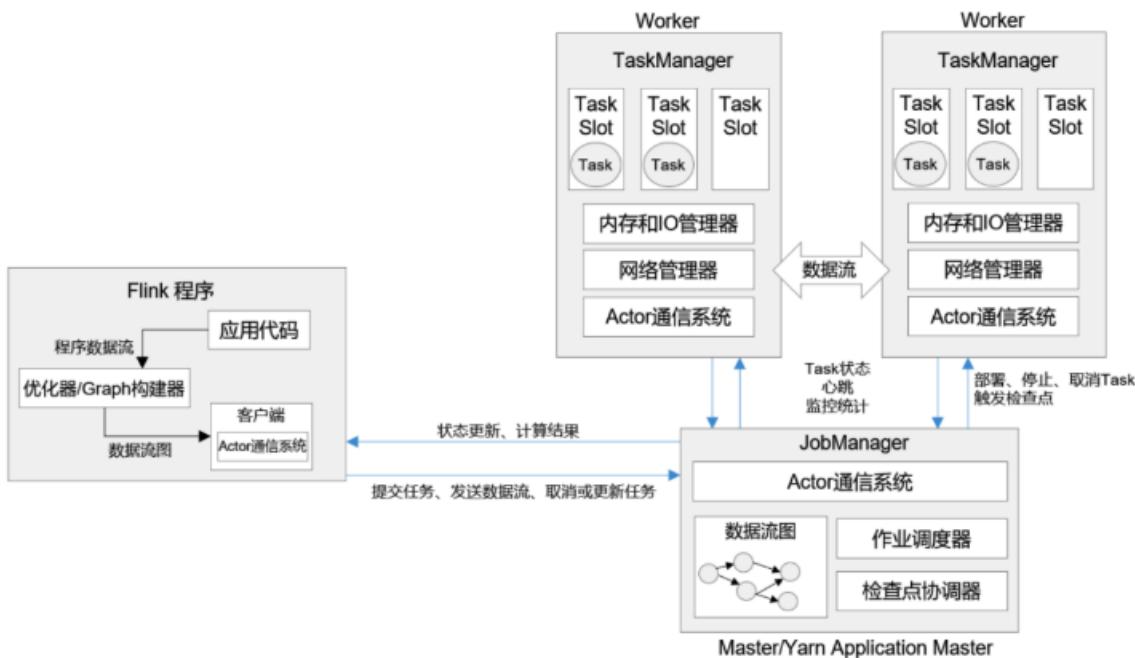
        //运行环境
        environment.execute();
    }
}

```

## 4. Flink 集群部署

- Flink的部署有三种模式，分别是Local, Standalone Cluster和Yarn Cluster

### 4.1. 环境搭建



| 节点名称   | JobManager | TaskManager |
|--------|------------|-------------|
| node01 | *          | *           |
| node01 |            | *           |
| node01 |            | *           |

- 更换JDK版本

- [root@123 ~]# rpm -ivh jdk-11.0.16.1\_linux-x64\_bin.rpm
- [root@123 ~]# vim /etc/profile
  - export JAVA\_HOME=/usr/java/jdk-11.0.16.1
- [root@123 ~]# source /etc/profile
- [root@123 ~]# rm -rf jdk-11.0.16.1\_linux-x64\_bin.rpm

- 安装Flink1.15

- 解压移动

- [root@node01 ~]# tar -zvxf flink-1.15.2-bin-scala\_2.12.tgz
- [root@node01 ~]# mv flink-1.15.2 /opt/yjx/
- [root@node01 ~]# rm -rf flink-1.15.2-bin-scala\_2.12.tgz
- [root@node01 ~]# cd /opt/yjx/flink-1.15.2/

- 集群配置

- [root@node01 flink-1.15.2]# vim conf/flink-conf.yaml

```

■ 31 # JobManager runs.
33 jobmanager.rpc.address: node01

47 jobmanager.bind-host: 0.0.0.0

64 taskmanager.bind-host: 0.0.0.0

66 # The address of the host on which the TaskManager runs and
can be reached by the JobManager and
  
```

```

67 # other TaskManagers. If not specified, the TaskManager will
try different strategies to identify
68 # the address.
76 taskmanager.host: node01

89 # The number of task slots that each TaskManager offers.
# Each slot runs one parallel pipeline.
91 taskmanager.numberOfTaskSlots: 2

93 # The parallelism used for programs that did not specify any
# other parallelism.
95 parallelism.default: 2

188 # The address to which the REST client will connect to
190 rest.address: node01

200 # To enable this, set the bind address to one that has
# access to outside-facing
201 # network interface, such as 0.0.0.0.
202 #
203 rest.bind-address: 0.0.0.0

```

- [root@node01 flink-1.15.2]# vim conf/masters

- node01:8081

- [root@node01 flink-1.15.2]# vim conf/workers

- node01
  - node02
  - node03

- 拷贝到其他节点

- [root@node02 ~]# scp -r root@node01:/opt/yjx/flink-1.15.2 /opt/yjx/
  - [root@node02 ~]# vim /opt/yjx/flink-1.15.2/conf/flink-conf.yaml

- 66 # The address of the host on which the TaskManager runs and can
be reached by the JobManager and
67 # other TaskManagers. If not specified, the TaskManager will try
different strategies to identify
68 # the address.
76 taskmanager.host: node02

- [root@node03 ~]# scp -r root@node01:/opt/yjx/flink-1.15.2 /opt/yjx/
  - [root@node03 ~]# vim /opt/yjx/flink-1.15.2/conf/flink-conf.yaml

- 66 # The address of the host on which the TaskManager runs and can
be reached by the JobManager and
67 # other TaskManagers. If not specified, the TaskManager will try
different strategies to identify
68 # the address.
76 taskmanager.host: node03

- 环境配置

- [root@123 ~]# vim /etc/profile

```
■ export FLINK_HOME=/opt/yjx/flink-1.15.2  
export PATH=$FLINK_HOME/bin:$PATH
```

- [root@123 ~]# source /etc/profile

- **开启集群**

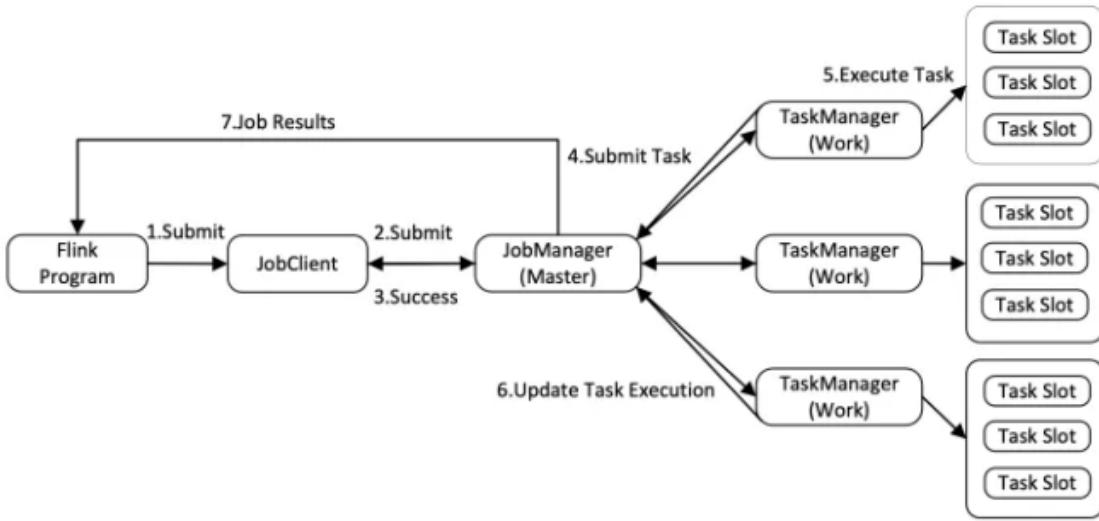
```
■ [root@node01 ~]# start-cluster.sh
```

```
[root@node01 ~]# start-cluster.sh  
Starting cluster.  
Starting standalonesession daemon on host node01.  
Warning: Permanently added 'node01,192.168.88.101' (ECDSA) to the list of known hosts.  
Starting taskexecutor daemon on host node01.  
Warning: Permanently added 'node02,192.168.88.102' (ECDSA) to the list of known hosts.  
Starting taskexecutor daemon on host node02.  
Warning: Permanently added 'node03,192.168.88.103' (ECDSA) to the list of known hosts.  
Starting taskexecutor daemon on host node03.
```

```
■ http://192.168.88.101:8081/
```

## 4.2. 系统架构

- Flink 的运行时架构中，最重要的就是两大组件：作业管理器（JobManager）和任务管理器（TaskManager）。
  - JobManager 是真正意义上的“管理者”（Master），负责管理调度，所以在不考虑高可用的情况下只能有一个；
  - TaskManager 是“工作者”（Worker、Slave），负责执行任务处理数据，所以可以有一个或多个。
- 客户端并不是处理系统的一部分，它只负责作业的提交。具体来说，就是调用程序的 main 方法，将代码转换成“数据流图”（Dataflow Graph），并最终生成作业图（JobGraph），一并发送给 JobManager。提交之后，任务的执行其实就跟客户端没有关系了；
-



#### 4.2.1. JobManager

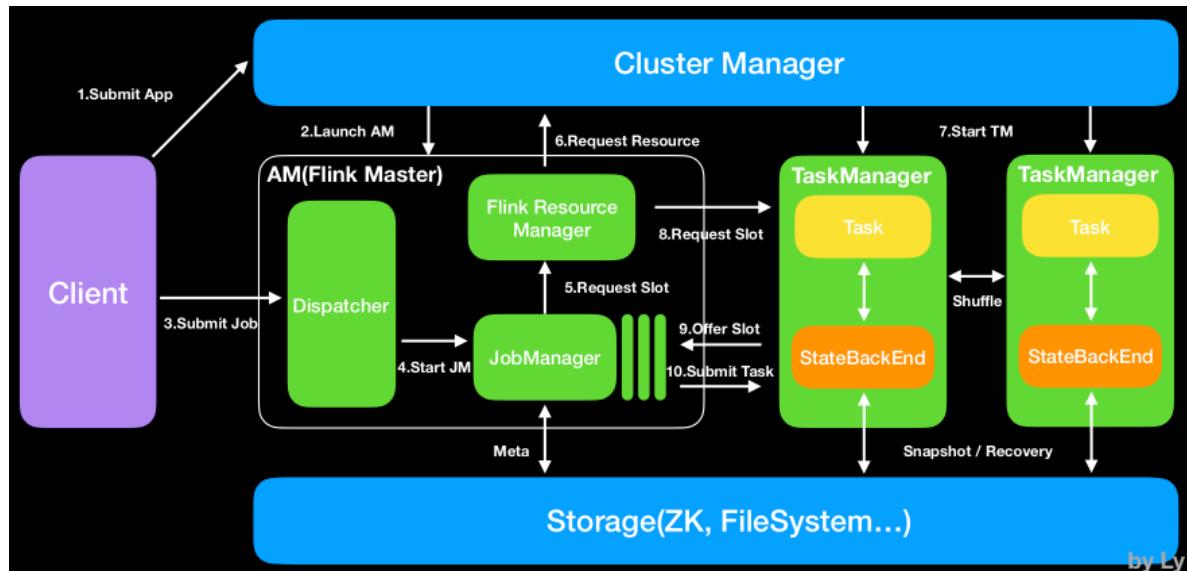
- JobManager 是一个 Flink 集群中任务管理和调度的核心，是控制应用执行的主进程。也就是说，每个应用都应该被唯一的 JobManager 所控制执行。JobManger 又包含 3 个不同的组件：
  - JobMaster
    - JobMaster 中最核心的组件，负责处理单独的作业 (Job)。所以 JobMaster 和具体的 Job 是一一对应的，多个 Job 可以同时运行在一个 Flink 集群中，每个 Job 都有一个自己的 JobMaster。
    - 在作业提交时，JobMaster 会先接收到要执行的应用。这里所说“应用”一般是客户端提交来的，包括：Jar 包，数据流图 (dataflow graph)，和作业图 (JobGraph)。
    - JobMaster 会把 JobGraph 转换成一个物理层面的数据流图，这个图被叫作“执行图” (ExecutionGraph)，它包含了所有可以并发执行的任务。JobMaster 会向资源管理器 (ResourceManager) 发出请求，申请执行任务必要的资源。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的 TaskManager 上。
  - ResourceManager
    - ResourceManager 主要负责资源的分配和管理，在 Flink 集群中只有一个。所谓“资源”，主要是指 。任务槽就是 Flink 集群中的资源调配单元，包含了机器用来执行计算的一组 CPU 和内存资源。每一个任务 (Task) 都需要分配到一个 slot 上执行。
    - 在 Standalone 部署时，因为 TaskManager 是单独启动的，所以 ResourceManager 只能分发可用 TaskManager 的任务槽，不能单独启动新 TaskManager。
    - 在其他资源管理平台时，ResourceManager 会将有空闲槽位的 TaskManager 分配给 JobMaster。如果 ResourceManager 没有足够的任务槽，它还可以向资源提供平台发起会话，请求提供启动 TaskManager 进程的容器。另外，ResourceManager 还负责停掉空闲的 TaskManager，释放计算资源。
  - Dispatcher
    - Dispatcher 主要负责提供一个 REST 接口，用来提交应用，并且负责为每一个新提交的作业启动一个新的 JobMaster 组件。Dispatcher 也会启动一个 Web UI，用来方便地展示和监控作业执行的信息。
    - Dispatcher 在架构中并不是必需的，在不同的部署模式下可能会被忽略掉。

#### 4.2.2. TaskManager

- TaskManager 是 Flink 中的工作进程，数据流的具体计算就是它来做的，所以也被称“Worker”。Flink 集群中必须至少有一个 TaskManager；当然由于分布式计算的考虑，通常会有多个 TaskManager 运行，每一个 TaskManager 都包含了一定数量的任务槽 (task slots)。Slot 是资源调度的最小单位，slot 的数量限制了 TaskManager 能够并行处理的任务数量。

- 启动之后，TaskManager 会向资源管理器注册它的 slots；收到资源管理器的指令后，TaskManager 就会将一个或者多个槽位提供给 JobMaster 调用，JobMaster 就可以分配任务来执行了。
- 在执行过程中，TaskManager 可以缓冲数据，还可以跟其他运行同一应用的 TaskManager 交换数据。

### 4.3. Standalone模式



#### 4.3.1. 理论分析

- standalone工作流程
  - 客户端不是运行时和程序执行的一部分，但它用于准备并发送dataflow(JobGraph)给 Master(JobManager)，然后，客户端断开连接或者维持连接以等待接收计算结果。
  - 当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。
  - Client 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。
  - JobManager 主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。
  - TaskManager 在启动的时候就设置好了槽位数 (Slot)，每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。
- standalone模式缺点
  - 资源利用弹性不够（资源总量是定死的；job 退出后也不能立刻回收资源）
  - 资源隔离度不够（所有 job 共享集群的资源）
  - 所有 job 共用一个 jobmanager，负载过大

#### 4.3.2. 任务提交

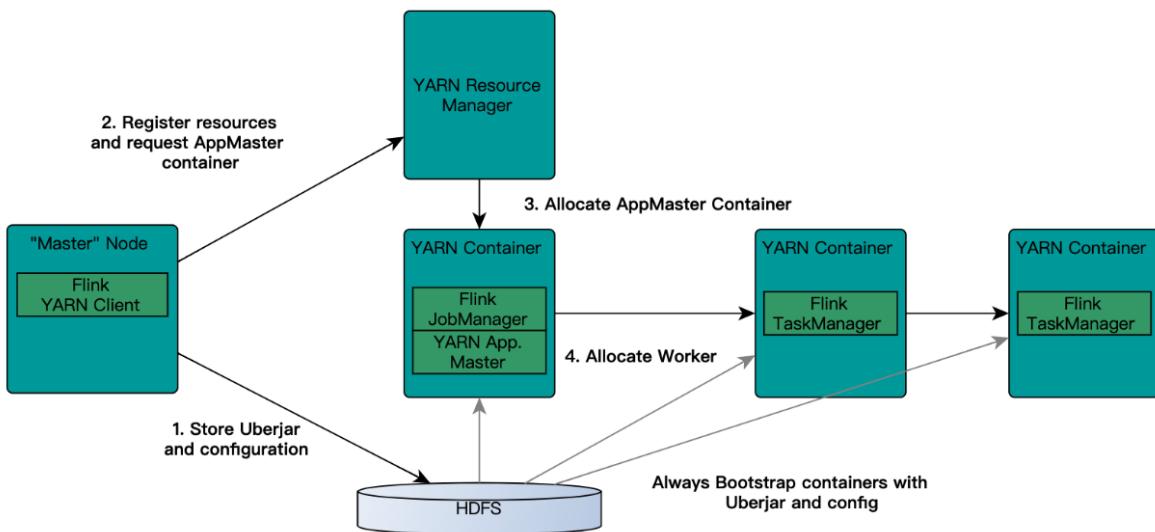
- 页面方式

| Uploaded Jars                                                         |                      |                                         |          |             |
|-----------------------------------------------------------------------|----------------------|-----------------------------------------|----------|-------------|
| Name                                                                  | Upload Time          | Entry Class                             |          |             |
| flink060106_util-1.0-SNAPSHOT.jar                                     | 2022-10-25, 16:11:53 | -                                       | Delete   |             |
| <input checked="" type="checkbox"/> com.yjxxt.flink.Hello01Standalone |                      |                                         |          |             |
| <input checked="" type="checkbox"/> Program Arguments                 |                      | <input type="checkbox"/> Savepoint Path |          |             |
| <input type="checkbox"/> Allow Non Restored State                     | Show Plan            | Submit                                  |          |             |
| Running Jobs                                                          |                      |                                         |          |             |
| Job Name                                                              | Start Time           | Duration                                | End Time | Tasks       |
| Flink Streaming Job                                                   | 2022-10-25 16:30:11  | 12s                                     | -        | 2 2 RUNNING |

- 命令提交

- [root@node01 ~]# flink run -c com.yjxxt.flink.Hello01Standalone -p 2 /root/flink060106\_util-1.0-SNAPSHOT.jar
  - c,-class : 需要指定的main方法的类
  - C,-classpath : 向每个用户代码添加url，他是通过UrlClassLoader加载。url需要指定文件的schema如 (file://)
  - d,-detached : 在后台运行
  - p,-parallelism : job需要指定env的并行度，这个一般都需要设置。
  - q,-sysoutLogging : 禁止logging输出作为标准输出。
  - s,-fromSavepoint : 基于savepoint保存下来的路径，进行恢复。
  - sae,-shutdownOnAttachedExit : 如果是前台的方式提交，当客户端中断，集群执行的job任务也会shutdown。

## 4.4. Yarn模式



### 4.4.1. 运行原理

- 步骤1：当启动一个新的 Flink YARN Client会话，客户端首先会检查所请求的资源（容器和内存）是否可用。之后，它会上传包含了 Flink 配置文件和 jar包到 HDFS.
- 步骤2：客户端的请求一个container资源去启动 ApplicationMaster 进程
- 步骤3：ResourceManager选一台NodeManager机器启动AM。
  - 注意1：
    - 在这过程中，因为客户端已经将配置文件和jar包作为容器的资源注册了，所以 NodeManager 会负责准备容器做一些初始化工作（例如，下载文件）。
    - 一旦这些完成了，ApplicationMaster (AM) 就启动了。
  - 注意2：
    - JobManager 和 AM 运行在同一个容器中。一旦它们成功地启动了，AM 知道 JobManager 的地址（它自己）。

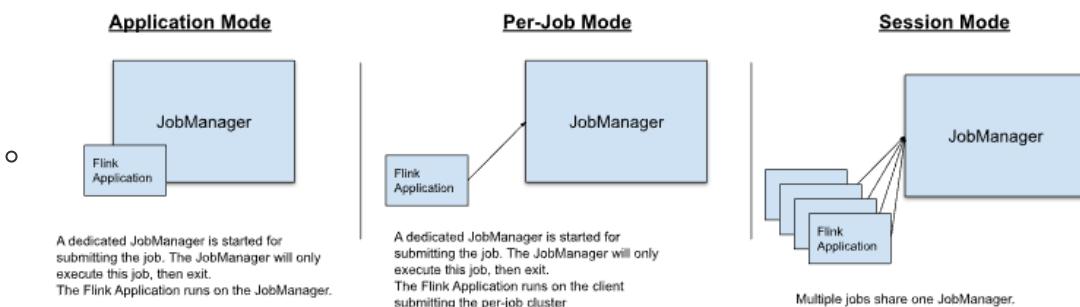
- 它会为 TaskManager 生成一个新的 Flink 配置文件（这样它们才能连上 JobManager）。该文件也同样会上传到 HDFS。
- 另外，AM 容器同时提供了 Flink 的 Web 界面服务。
- 步骤4：AM 开始为 Flink 的 TaskManager 分配容器(container)，在对应的nodemanager上面启动taskmanager.
- 步骤5：初始化工作，从 HDFS 下载 jar 文件和修改过的配置文件。一旦这些步骤完成了，Flink 就安装完成并准备接受任务了。

#### 4.4.2. 集成环境

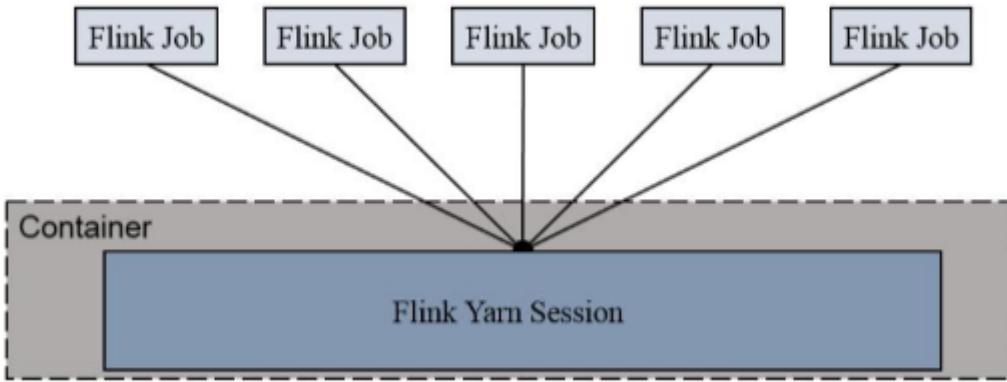
- [root@123 ~]# vim /etc/profile
  - export HADOOP\_CONF\_DIR=/opt/yjx/hadoop-3.1.2/etc/hadoop/
- [root@123 ~]# source /etc/profile
- 上传Flink与Hadoop的连接包
  - flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar
  - commons-cli-1.4.jar
- 拷贝到其他节点
  - [root@123 ~]# scp root@node01:/root/flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar /opt/yjx/flink-1.15.2/lib/
  - [root@123 ~]# scp root@node01:/root/commons-cli-1.4.jar /opt/yjx/flink-1.15.2/lib/

#### 4.4.3. 三种模式

- Flink可以通过以下三种方式之一执行应用程序：
  - in Application Mode,
  - in a Per-Job Mode,
  - in Session Mode.
- 上述模式的不同之处在于：
  - 集群生命周期和资源隔离保证
  - 应用程序的main方法是在客户端上还是在集群上执行。



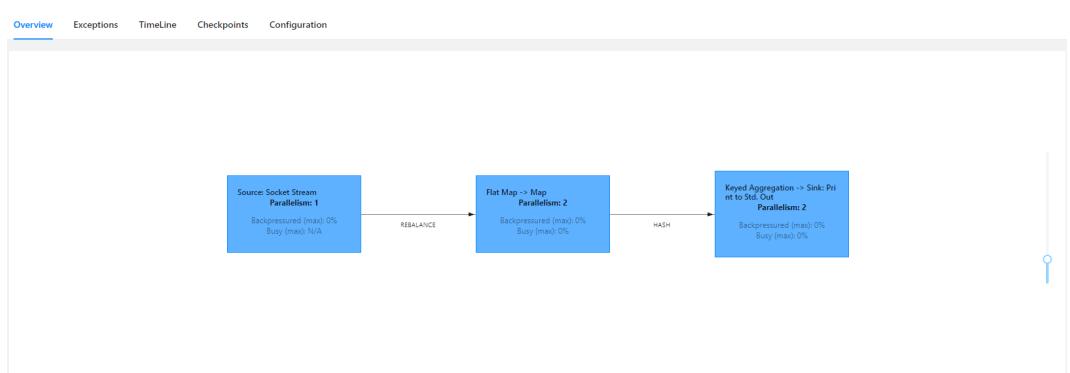
#### 4.4.4. Session Mode



- 多个 job 共享同一个集群<jobmanager/taskmanager>、job 退出集群也不会退出，用户类的 main 方法在client 端运行；
- 需要频繁提交大量小 job 的场景比较适用；因为每次提交一个新 job 的时候，不需要去向 yarn 注册应用
- 特点：需要事先申请资源，启动JobManager 和 TaskManager
  - 优点：不需要每次递交作业申请资源，使用已有资源
  - 缺点：作业执行完成后，资源不会被释放，因此会一直占用系统资源
- 使用场景：适合小作业比较多，作业递交比较频繁的场景
- 启动yarn-session并分配的资源：
  - yarn-session.sh -n 3 -jm 1024 -tm 1024
  - -n 指明container容器个数，即 taskmanager的进程个数。
  - -jm 指明jobmanager进程的内存大小
  - -tm 指明每个taskmanager的进程内存大小

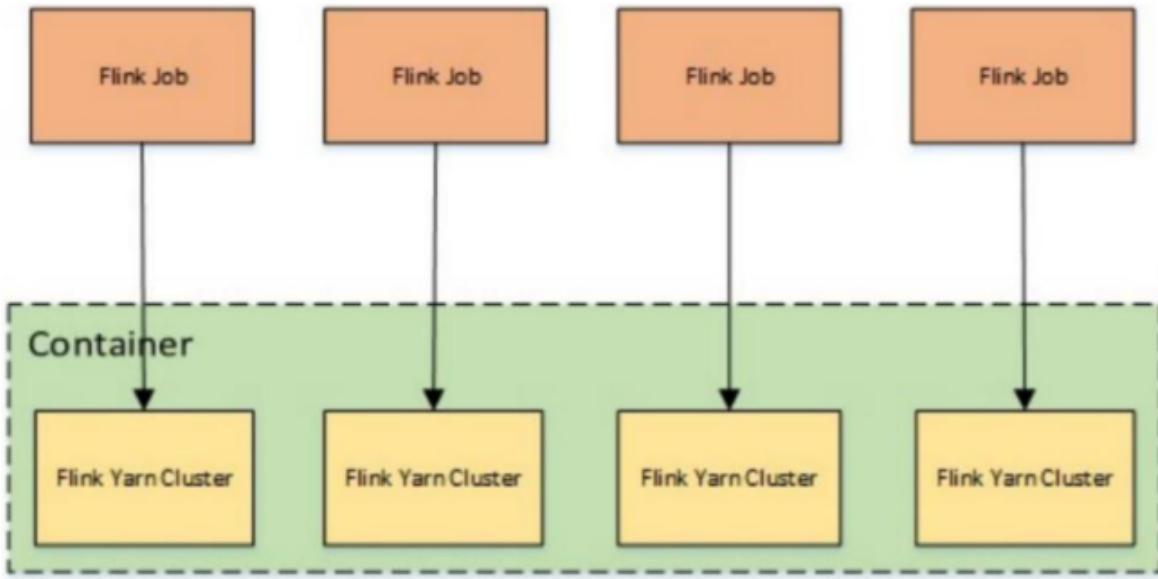
```
2022-10-25 18:08:38,801 INFO org.apache.flink.yarn.YarnClusterDescriptor []| Submitting application master application_1666689812613_0004
2022-10-25 18:08:38,801 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl []| Submitted application application_1666689812613_0004
2022-10-25 18:08:38,843 INFO org.apache.flink.yarn.YarnClusterDescriptor []| Waiting for the cluster to be allocated
2022-10-25 18:08:38,544 INFO org.apache.flink.yarn.YarnClusterDescriptor []| Deploying cluster, current state ACCEPTED
2022-10-25 18:08:49,618 INFO org.apache.flink.yarn.YarnClusterDescriptor []| YARN application has been deployed successfully.
2022-10-25 18:08:49,619 INFO org.apache.flink.yarn.YarnClusterDescriptor []| Found Web Interface node03:39495 of application 'application_1666689812613_0004'.
JobManager Web Interface: http://node03:39495
```

- 提交任务
    - JobManager Web Interface: <http://node03:39495>
    - flink run -c com.yjxxt.flink.Hello01Standalone -p 2 -m node03:39495 /root/flink060106\_util.jar
- ```
[root@node01 ~]# flink run -c com.yjxxt.flink.Hello01Standalone -p 2 -m node03:39495 /root/flink060106_util.jar
2022-10-25 18:11:48,021 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli []| - Found Yarn properties file under /tmp/.yarn-properties-root.
2022-10-25 18:11:48,021 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli []| - Found Yarn properties file under /tmp/.yarn-properties-root.
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.flink.api.java.ClosureCleaner (file:/opt/yjx/flink-1.15.2/lib/flink-dist-1.15.2.jar) to field java.lang.String.value
WARNING: Please consider reporting this to the maintainers of org.apache.flink.api.java.ClosureCleaner
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Job has been submitted with JobID 280155d93d15bb7e62b7ff03d13996498
application_1666689812613_0004 root Flink session Flink default 0 Tue Oct 25 N/A RUNNING UNDEFINED 2 2 4096 0 0 16.7 16.7 [Application]
点击查看任务
```



- 停止yarn上的flink集群：
  - yarn application -kill application\_id
  - [root@node01 ~]# yarn application -kill application\_1666689812613\_0004
 Killing application application\_1666689812613\_0004
 2022-10-25 18:16:02,190 INFO impl.YarnClientImpl: Killed application application\_1666689812613\_0004

#### 4.4.5. Per-Job Mode



- 每个 job 独享一个集群，job 退出集群则退出，用户类的 main 方法在 client 端运行；
  - 比较合适大 job，运行时长很长；因为每起一个 job，都要去向 yarn 申请容器启动 jm,tm，比较耗时
  - 一个任务会对应一个 Job，每提交一个作业会根据自身的情况，向都会单独向 yarn 申请资源，直到作业执行完成，一个作业的失败与否并不会影响下一个作业的正常提交和运行。独享 Dispatcher 和 ResourceManager，按需接受资源申请；
  - 特点：不需要事先申请资源，在每次递交作业的时候申请一次资源
    - 优点：作业运行完成资源立刻会被释放
    - 缺点：每次递交作业都需要申请资源，会影响执行效率，申请资源需要消耗时间
  - 使用场景：适合大作业，作业比较少的场景
  - 提交任务：
    - flink run -m yarn-cluster -yjm 1024 -ytm 1024 -c com.yjxxt.flink.Hello01StandAlone flink060105\_tools-1.0-SNAPSHOT.jar
    - -m: 后面跟的是yarn-cluster，不需要指明地址。这是由于Single job模式是每次提交任务会新建flink集群，所以它的jobmanager是不固定的
    - -yn: 指明taskmanager个数。
- ```
root@node01:~$ flink run -m yarn-cluster -yjm 1024 -ytm 1024 /root/flink060106_util.jar
WARNING: An illegal reflective access occurs has occurred
WARNING: Illegal reflective access by org.apache.flink.api.java.ClosureCleaner (file:/opt/yjx/flink-1.15.2/lib/flink-dist-1.15.2.jar) to field java.lang.String.value
WARNING: Please consider reporting this to the maintainers of org.apache.flink.api.java.ClosureCleaner
WARNING: It seems likely that multiple classes have reflective access operations
WARNING: All illegal access operations will be denied in a future release
2022-10-25 18:17:49,590 WARN  org.apache.flink.yarn.configuration.YarnLogConfigUtil      [] - The configuration directory ('/opt/yjx/flink-1.15.2/conf') already contains a LOG4J config file. If you want to use logback, then please delete or rename the log configuration file.
2022-10-25 18:17:49,742 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - No path for the flink jar passed. Using the location of class org.apache.flink.yarn.YarnClusterDescriptor to locate the jar
2022-10-25 18:17:49,763 WARN  org.apache.flink.yarn.YarnClusterDescriptor          [] - Job Clusters are deprecated since Flink 1.15. Please use an Application Cluster/Application Mode instead.
2022-10-25 18:17:49,919 INFO  org.apache.hadoop.conf.Configuration           [] - resource-types.xml not found
2022-10-25 18:17:49,919 INFO  org.apache.hadoop.yarn.util.resource.ResourceUtils      [] - Unable to find 'resource-types.xml'.
2022-10-25 18:17:49,958 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - Cluster specification: ClusterSpecification(masterMemoryMB=1024, taskManagerMemoryMB=1024, slotPerTaskManager=2)
2022-10-25 18:18:19,550 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - Submitting application master application_166669812613_0005
2022-10-25 18:18:20,183 INFO  org.apache.hadoop.yarn.client.api.impl.YarnClientImpl     [] - Submitted application application_166669812613_0005
2022-10-25 18:18:20,183 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - Waiting for the cluster to be allocated
2022-10-25 18:18:43,723 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - Deploying cluster, current state ACCEPTED
2022-10-25 18:18:43,723 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - YARN application has been deployed successfully.
2022-10-25 18:18:43,723 INFO  org.apache.flink.yarn.YarnClusterDescriptor          [] - Found Web Interface node03:39466 of application 'application_166669812613_0005'.
Job has been submitted with JobID 0ad35d5e195058c43e6230488e0ff451
```

#### 4.4.6. Application Mode

- 应用模式的特点和分离模式很像，区别在于：
- 用户程序的 main 方法在集群中运行，而不是在客户端运行
- 应用模式为每个提交的应用程序创建一个集群，该集群可以看作是在特定应用程序的作业之间共享的会话集群，并在应用程序完成时终止。在这种体系结构中，应用模式在不同应用之间提供了资源隔离和负载平衡保证

- 用户可以手动将应用程序jar及依赖的jar事先上传到hdfs，然后每次递交作业的时候不需要上传jar了，只需要指定hdfs已上传的jar路径即可

- #提交任务

```
flink run-application -t yarn-application /root/flink060106_util.jar
```

#列出集群上正在运行的作业，列出jobId、jobName

```
flink list -t yarn-application -  
Dyarn.application.id=application_1666689812613_0003
```

#取消任务： jobId 【请注意， 取消应用程序集群上的作业将停止该集群。】

```
flink cancel -t yarn-application -  
Dyarn.application.id=application_1666689812613_0003
```

The screenshot shows a table of applications. There is one entry:

| ID                             | User | Name                      | Application Type | Queue   | Application Priority | StartTime                         | FinishTime | State   | FinalStatus | Running Containers | Allocated CPU VCores | Allocated Memory MB | Reserved CPU VCores | Reserved Memory MB | % of Queue | % of Cluster | Progress                    | Tracking URL |
|--------------------------------|------|---------------------------|------------------|---------|----------------------|-----------------------------------|------------|---------|-------------|--------------------|----------------------|---------------------|---------------------|--------------------|------------|--------------|-----------------------------|--------------|
| application_1666689812613_0003 | root | Flink Application Cluster | Apache Flink     | default | 0                    | Tue Oct 25<br>17:54:46 +0800 2022 | N/A        | RUNNING | UNDEFINED   | 2                  | 2                    | 4096                | 0                   | 0                  | 16.7       | 16.7         | <a href="#">Application</a> |              |

[点击跳转到Flink任务页面](#)

## 5. Flink CEP

小宝贝 你最美 早点睡 喝热水 我爱你 别误会 那个女孩是我妹  
你干啥 真没有 我们只是喝点酒 喝多了 拉拉手 大家都是好朋友  
别闹了 对不起 反正都是你有理 你很好 我不配 忘了我吧下一位  
吃了没 早点睡 这个东西有点贵 没电了 刚开机 刚才一直在想你  
我有事 刚忙完 忙了一天我很累 我爱你 怎么会 那个女孩是我妹  
你别闹 真没有 就是一起喝点酒 喝多了 乱说的 我们只是好朋友

### 5.1. 概念

- 概念
  - CEP(Complex Event Processing)
  - FlinkCEP是在Flink上层实现的复杂事件处理库。 它可以让你在无限事件流中检测出特定的事件模型，有机会掌握数据中重要的那部分。
  - 市场上有多重 CEP 的解决方案，例如 Spark、Samza、Beam 等，但他们都没有提供专门的 library 支持。Flink 提供了专门的 CEP library。
- 特征：
  - 目标：从有序的简单事件流中发现一些高阶特征
  - 输入：一个或多个由简单事件构成的事件流
  - 处理：识别简单事件之间的内在联系，多个符合一定规则的简单事件构成复杂事件
  - 输出：满足规则的复杂事件
- maven

- ```

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep</artifactId>
    <version>${flink.version}</version>
</dependency>

```
- ```

DataStream<Event> input = ...;

Pattern<Event, ?> pattern = Pattern.<Event>begin("start").where(
    new SimpleCondition<Event>() {
        @Override
        public boolean filter(Event event) {
            return event.getId() == 42;
        }
    }
).next("middle").subtype(SubEvent.class).where(
    new SimpleCondition<SubEvent>() {
        @Override
        public boolean filter(SubEvent subEvent) {
            return subEvent.getVolume() >= 10.0;
        }
}
).followedBy("end").where(
    new SimpleCondition<Event>() {
        @Override
        public boolean filter(Event event) {
            return event.getName().equals("end");
        }
    }
);

```

```

PatternStream<Event> patternStream = CEP.pattern(input, pattern);

DataStream<Alert> result = patternStream.process(
    new PatternProcessFunction<Event, Alert>() {
        @Override
        public void processMatch(
            Map<String, List<Event>> pattern,
            Context ctx,
            Collector<Alert> out) throws Exception {
            out.collect(createAlertFrom(pattern));
        }
    });

```

## 5.2. 模式

- 模式API可以让你定义想从输入流中抽取的复杂模式序列
- 每个复杂的模式序列包括多个简单的模式，比如，寻找拥有相同属性事件序列的模式。
- 从现在开始，我们把这些简单的模式称作**模式**，把我们在数据流中最终寻找的复杂模式序列称作**模式序列**，你可以把模式序列看作是这样的模式构成的图，这些模式基于用户指定的**条件**从一个转换到另外一个，比如 `event.getName().equals("end")`。

- 一个**匹配**是输入事件的一个序列，这些事件通过一系列有效的模式转换，能够访问到复杂模式图中的所有模式。
- 注意：
  - 每个模式必须有一个独一无二的名字，你可以在后面使用它来识别匹配到的事件。
  - 模式的名字不能包含字符 ":"。

### 5.2.1. 单个模式

- 一个模式可以是一个单例或者循环模式。单例模式只接受一个事件，循环模式可以接受多个事件。
- 在模式匹配表达式中，模式 "a b+ c? d" (或者 "a"，后面跟着一个或者多个 "b"，再往后可选择的跟着一个 "c"，最后跟着一个 "d")
  - a, c?, 和 d 都是单例模式，
  - b+ 是一个循环模式。
- 默认情况下，模式都是单例的，你可以通过使用量词把它们转换成循环模式。
- 量词
  - 在 FlinkCEP 中，循环模式：
    - `pattern.oneOrMore()`，指定期望一个给定事件出现一次或者多次的模式（例如前面提到的 `b+` 模式）；
    - `pattern.times(#ofTimes)`，指定期望一个给定事件出现特定次数的模式，例如出现 4 次 `a`；
    - `pattern.times(#fromTimes, #toTimes)`，指定期望一个给定事件出现次数在一个最小值和最大值中间的模式，比如出现 2-4 次 `a`。
    - `pattern.greedy()` 方法让循环模式变成贪心的，但现在还不能让模式组贪心。你可以使用 `pattern.optional()` 方法让所有的模式变成可选的，不管是否是循环模式。
  - 参考代码

```

// 期望出现4次
start.times(4);

// 期望出现0或者4次
start.times(4).optional();

// 期望出现2、3或者4次
start.times(2, 4);

// 期望出现2、3或者4次，并且尽可能的重复次数多
start.times(2, 4).greedy();

// 期望出现0、2、3或者4次
start.times(2, 4).optional();

// 期望出现0、2、3或者4次，并且尽可能的重复次数多
start.times(2, 4).optional().greedy();

// 期望出现1到多次
start.oneOrMore();

// 期望出现1到多次，并且尽可能的重复次数多
start.oneOrMore().greedy();

// 期望出现0到多次
start.oneOrMore().optional();

```

```

// 期望出现0到多次，并且尽可能的重复次数多
start.oneOrMore().optional().greedy();

// 期望出现2到多次
start.timesOrMore(2);

// 期望出现2到多次，并且尽可能的重复次数多
start.timesOrMore(2).greedy();

// 期望出现0、2或多次
start.timesOrMore(2).optional();

// 期望出现0、2或多次，并且尽可能的重复次数多
start.timesOrMore(2).optional().greedy();

```

- 条件

- 对每个模式你可以指定一个条件来决定一个进来的事件是否被接受进入这个模式，例如，它的value字段应该大于5，或者大于前面接受的事件的平均值。
- 指定判断事件属性的条件可以通过 `pattern.where()`、`pattern.or()` 或者 `pattern.until()` 方法。
- 这些可以是 `IterativeCondition` 或者 `simpleCondition`。
- **简单条件:**
  - 这种类型的条件扩展了前面提到的 `IterativeCondition` 类，它决定是否接受一个事件只取决于事件自身的属性。
- **迭代条件:**
  - 这是最普遍的条件类型。使用它可以指定一个基于前面已经被接受的事件的属性或者它们的一个子集的统计数据来决定是否接受时间序列的条件。
- **组合条件:**
  - 你可以把 `subtype` 条件和其他的条件结合起来使用。这适用于任何条件，你可以通过依次调用 `where()` 来组合条件。最终的结果是每个单一条件的结果的逻辑**AND**。如果想使用**OR**来组合条件，你可以像下面这样使用 `or()` 方法。
- **停止条件:**
  - 如果使用循环模式（`oneOrMore()` 和 `oneOrMore().optional()`），你可以指定一个停止条件，例如，接受事件的值大于5直到值的和小于50。
- **具体实现**
  - `where(condition)`
    - 为了匹配这个模式，一个事件必须满足某些条件。多个连续的 `where()` 语句取与组成判断条件。
  - `or(condition)`
    - 增加一个新的判断，和当前的判断取或。一个事件只要满足至少一个判断条件就匹配到模式。
  - `until(condition)`
    - 为循环模式指定一个停止条件。意思是满足了给定的条件的事件出现后，就不会再有事件被接受进入模式了。只适用于和`oneOrMore()`同时使用。
  - `subtype(subClass)`
    - 为当前模式定义一个子类型条件。一个事件只有是这个子类型的时候才能匹配到模式。

- `oneOrMore()`
  - 指定模式期望匹配到的事件至少出现一次。
- `timesOrMore(#times)`
  - 指定模式期望匹配到的事件至少出现 #times 次。
- `times(#ofTimes)`
  - 指定模式期望匹配到的事件正好出现的次数。
- `times(#fromTimes, #toTimes)`
  - 指定模式期望匹配到的事件出现次数在#fromTimes和#toTimes之间。
- `optional()`
  - 指定这个模式是可选的，也就是说，它可能根本不出现。这对所有之前提到的量词都适用。
- `greedy()`
  - 指定这个模式是贪心的，也就是说，它会重复尽可能多的次数。

## 5.2.2. 组合模式

- 模式序列由一个初始模式作为开头
  - `Pattern<Event, ?> start = Pattern.<Event>begin("start");`
- 增加更多的模式到模式序列中并指定它们之间所需的连续条件。
  - **严格连续**: 期望所有匹配的事件严格的一个接一个出现，中间没有任何不匹配的事件。
  - **松散连续**: 忽略匹配的事件之间的不匹配的事件。
  - **不确定的松散连续**: 更进一步的松散连续，允许忽略掉一些匹配事件的附加匹配。
- 连续条件代码实现
  - `next()`, 指定**严格连续**,
  - `followedBy()`, 指定**松散连续**,
  - `followedByAny()`, 指定**不确定的松散连续**。
  - `notNext()`, 如果不想后面直接连着一个特定事件
  - `notFollowedBy()`, 如果不想一个特定事件发生在两个事件之间的任何地方。
  - ```
// 严格连续
Pattern<Event, ?> strict = start.next("middle").where(...);

// 松散连续
Pattern<Event, ?> relaxed = start.followedBy("middle").where(...);

// 不确定的松散连续
Pattern<Event, ?> nonDetermin = 
start.followedByAny("middle").where(...);

// 严格连续的NOT模式
Pattern<Event, ?> strictNot = start.notNext("not").where(...);

// 松散连续的NOT模式
Pattern<Event, ?> relaxedNot = start.notFollowedBy("not").where(...);
```
- 注意

- 一个模式序列只能有一个时间限制。如果限制了多个时间在不同的单个模式上，会使用最小的那个时间限制。
  - next.within(Time.seconds(10));
  - 模式序列不能以 `notFollowedBy()` 结尾。
  - 一个 `NOT` 模式前面不能是可选的模式。
- 循环模式中的连续性
  - 数据: "a", "b1", "d1", "b2", "d2", "b3", "c"
  - 模式: "a b+ c"
  - **严格连续**: {a b1 c}, {a b2 c}, {a b3 c} - 没有相邻的 "b"。
  - **松散连续**: {a b1 c}, {a b1 b2 c}, {a b1 b2 b3 c}, {a b2 c}, {a b2 b3 c}, {a b3 c} - "d" 都被忽略了。
  - **不确定松散连续**: {a b1 c}, {a b1 b2 c}, {a b1 b3 c}, {a b1 b2 b3 c}, {a b2 c}, {a b2 b3 c}, {a b3 c}
- consecutive
  - 循环模式 (例如 `oneOrMore()` 和 `times()`)，默认是松散连续。
  - 如果想使用严格连续，需要使用 `consecutive()` 方法明确指定，如果想使用不确定松散连续，可以使用 `allowCombinations()` 方法。

### 5.2.3. 模式组

- 可以定义一个模式序列作为 `begin`, `followedBy`, `followedByAny` 和 `next` 的条件。
- 这个模式序列在逻辑上会被当作匹配的条件，并且返回一个 `GroupPattern`，可以在 `GroupPattern` 上使用 `oneOrMore()`, `times(#ofTimes)`, `times(#fromTimes, #toTimes)`, `optional()`, `consecutive()`, `allowCombinations()`。
- ```

    Pattern<Event, ?> start = Pattern.begin(
        Pattern.
        <Event>begin("start").where(...).followedBy("start_middle").where(...)
    );

    // 严格连续
    Pattern<Event, ?> strict = start.next(
        Pattern.
        <Event>begin("next_start").where(...).followedBy("next_middle").where(...)
    ).times(3);

    // 松散连续
    Pattern<Event, ?> relaxed = start.followedBy(
        Pattern.
        <Event>begin("followedby_start").where(...).followedBy("followedby_middle").
        where(...).
        oneOrMore();

    // 不确定松散连续
    Pattern<Event, ?> nonDetermin = start.followedByAny(
        Pattern.
        <Event>begin("followedbyany_start").where(...).followedBy("followedbyany_middle").
        where(...).
        optional();
    
```

- 模式操作

| 模式操作                                          | 描述                                                                                                                                                                                                             |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>begin(#name)</code>                     | 定义一个开始的模式: <code>java Pattern start = Pattern.begin("start");</code>                                                                                                                                           |
| <code>begin(#pattern_sequence)</code>         | 定义一个开始的模式: <code>java Pattern start = Pattern.begin(Pattern.begin("start").where(...).followedBy("middle").where(...));</code>                                                                                 |
| <code>next(#name)</code>                      | 增加一个新的模式。匹配的事件必须是直接跟在前面匹配到的事件后面 (严格连续) : <code>java Pattern next = start.next("middle");</code>                                                                                                                |
| <code>next(#pattern_sequence)</code>          | 增加一个新的模式。匹配的事件序列必须是直接跟在前面匹配到的事件后面 (严格连续) : <code>java Pattern next = start.next(Pattern.begin("start").where(...).followedBy("middle").where(...));</code>                                                     |
| <code>followedBy(#name)</code>                | 增加一个新的模式。可以有其他事件出现在匹配的事件和之前匹配到的事件中间 (松散连续) : <code>java Pattern followedBy = start.followedBy("middle");</code>                                                                                                |
| <code>followedBy(#pattern_sequence)</code>    | 增加一个新的模式。可以有其他事件出现在匹配的事件序列和之前匹配到的事件中间 (松散连续) : <code>java Pattern followedBy = start.followedBy(Pattern.begin("start").where(...).followedBy("middle").where(...));</code>                                     |
| <code>followedByAny(#name)</code>             | 增加一个新的模式。可以有其他事件出现在匹配的事件和之前匹配到的事件中间, 每个可选的匹配事件都会作为可选的匹配结果输出 (不确定的松散连续) : <code>java Pattern followedByAny = start.followedByAny("middle");</code>                                                              |
| <code>followedByAny(#pattern_sequence)</code> | 增加一个新的模式。可以有其他事件出现在匹配的事件序列和之前匹配到的事件中间, 每个可选的匹配事件序列都会作为可选的匹配结果输出 (不确定的松散连续) : <code>java Pattern followedByAny = start.followedByAny(Pattern.begin("start").where(...).followedBy("middle").where(...));</code> |
| <code>notNext()</code>                        | 增加一个新的否定模式。匹配的 (否定) 事件必须直接跟在前面匹配到的事件之后 (严格连续) 来丢弃这些部分匹配: <code>java Pattern notNext = start.notNext("not");</code>                                                                                             |
| <code>notFollowedBy()</code>                  | 增加一个新的否定模式。即使有其他事件在匹配的 (否定) 事件和之前匹配的事件之间发生, 部分匹配的事件序列也会被丢弃 (松散连续) : <code>java Pattern notFollowedBy = start.notFollowedBy("not");</code>                                                                      |
| <code>within(time)</code>                     | 定义匹配模式的事件序列出现的最大时间间隔。如果未完成的事件序列超过了这个事件, 就会被丢弃: <code>java pattern.within(Time.seconds(10));</code>                                                                                                             |

#### 5.2.4. 匹配后跳过策略

- 对于一个给定的模式, 同一个事件可能会分配到多个成功的匹配上。
- 为了控制一个事件会分配到多少个匹配上, 你需要指定跳过策略 `AfterMatchSkipStrategy`。
- 有五种跳过策略, 如下:
  - NO\_SKIP**: 每个成功的匹配都会被输出。
  - SKIP\_TO\_NEXT**: 丢弃以相同事件开始的所有部分匹配。
  - SKIP\_PAST\_LAST\_EVENT**: 丢弃起始在这个匹配的开始和结束之间的所有部分匹配。
  - SKIP\_TO\_FIRST**: 丢弃起始在这个匹配的开始和第一个出现的名称为`PatternName`事件之间的所有部分匹配。
  - SKIP\_TO\_LAST**: 丢弃起始在这个匹配的开始和最后一个出现的名称为`PatternName`事件之间的所有部分匹配。
- 案例01: 给定一个模式 `b+ c` 和一个数据流 `b1 b2 b3 c`, 不同跳过策略之间的不同如下:

- | 跳过策略                        | 结果                               | 描述                                                             |
|-----------------------------|----------------------------------|----------------------------------------------------------------|
| <b>NO_SKIP</b>              | b1 b2<br>b3 c<br>b2 b3<br>c b3 c | 找到匹配 b1 b2 b3 c 之后，不会丢弃任何结果。                                   |
| <b>SKIP_TO_NEXT</b>         | b1 b2<br>b3 c<br>b2 b3<br>c b3 c | 找到匹配 b1 b2 b3 c 之后，不会丢弃任何结果，因为没有以 b1 开始的其他匹配。                  |
| <b>SKIP_PAST_LAST_EVENT</b> | b1 b2<br>b3 c                    | 找到匹配 b1 b2 b3 c 之后，会丢弃其他所有的部分匹配。                               |
| <b>SKIP_TO_FIRST[ b ]</b>   | b1 b2<br>b3 c<br>b2 b3<br>c b3 c | 找到匹配 b1 b2 b3 c 之后，会尝试丢弃所有在 b1 之前开始的部分匹配，但没有这样的匹配，所以没有任何匹配被丢弃。 |
| <b>SKIP_TO_LAST[ b ]</b>    | b1 b2<br>b3 c<br>b3 c            | 找到匹配 b1 b2 b3 c 之后，会尝试丢弃所有在 b3 之前开始的部分匹配，有一个这样的 b2 b3 c 被丢弃。   |

- 案例02：模式：(a | b | c) (b | c) c+.greedy d，输入：a b c1 c2 c3 d，结果将会是：

- | 跳过策略                       | 结果                                           | 描述                                                                  |
|----------------------------|----------------------------------------------|---------------------------------------------------------------------|
| <b>NO_SKIP</b>             | a b c1 c2 c3<br>d b c1 c2 c3<br>d c1 c2 c3 d | 找到匹配 a b c1 c2 c3 d 之后，不会丢弃任何结果。                                    |
| <b>SKIP_TO_FIRST[ c* ]</b> | a b c1 c2 c3<br>d c1 c2 c3 d                 | 找到匹配 a b c1 c2 c3 d 之后，会丢弃所有在 c1 之前开始的部分匹配，有一个这样的 b c1 c2 c3 d 被丢弃。 |

- 案例03：模式：a b+，输入：a b1 b2 b3，结果将会是：

- | 跳过策略                | 结果                            | 描述                                                             |
|---------------------|-------------------------------|----------------------------------------------------------------|
| <b>NO_SKIP</b>      | a b1 a b1<br>b2 a b1<br>b2 b3 | 找到匹配 a b1 之后，不会丢弃任何结果。                                         |
| <b>SKIP_TO_NEXT</b> | a b1                          | 找到匹配 a b1 之后，会丢弃所有以 a 开始的部分匹配。这意味着不会产生 a b1 b2 和 a b1 b2 b3 了。 |

- 想指定要使用的跳过策略，只需要调用下面的方法创建 AfterMatchSkipStrategy：

| 方法                                              | 描述                                    |
|-------------------------------------------------|---------------------------------------|
| AfterMatchskipStrategy.noSkip()                 | 创建NO_SKIP策略                           |
| AfterMatchskipStrategy.skipToNext()             | 创建SKIP_TO_NEXT策略                      |
| AfterMatchskipStrategy.skipPastLastEvent()      | 创建SKIP_PAST_LAST_EVENT策略              |
| AfterMatchskipStrategy.skipToFirst(patternName) | 创建引用模式名称为patternName的 SKIP_TO_FIRST策略 |
| AfterMatchskipStrategy.skipToLast(patternName)  | 创建引用模式名称为patternName的 SKIP_TO_LAST策略  |

- AfterMatchskipStrategy skipStrategy = ...;
 Pattern.begin("patternName", skipStrategy);

### 5.3. 检测模式

- 在指定了要寻找的模式后，该把它们应用到输入流上来发现可能的匹配了。为了在事件流上运行你的模式，需要创建一个PatternStream。给定一个输入流 input，一个模式 pattern 和一个可选的用来对使用事件时间时有同样时间戳或者同时到达的事件进行排序的比较器 comparator。

```

o DataStream<Event> input = ...
Pattern<Event, ?> pattern = ...
EventComparator<Event> comparator = ... // 可选的

PatternStream<Event> patternStream = CEP.pattern(input, pattern,
comparator);

```

- 在获得到一个PatternStream之后，你可以应用各种转换来发现事件序列。推荐使用PatternProcessFunction。
- 匹配数据操作
  - PatternProcessFunction有一个processMatch的方法在每找到一个匹配的事件序列时都会被调用。它按照Map<String, List<IN>>的格式接收一个匹配，映射的键是你的模式序列中的每个模式的名称，值是被接受的事件列表（IN是输入事件的类型）。模式的输入事件按照时间戳进行排序。为每个模式返回一个接受的事件列表的原因是当使用循环模式（比如oneToMany()和times()）时，对一个模式会有不止一个事件被接受。

```

o class MyPatternProcessFunction<IN, OUT> extends
  PatternProcessFunction<IN, OUT> {
  @Override
  public void processMatch(Map<String, List<IN>> match, Context ctx,
  Collector<OUT> out) throws Exception {
    IN startEvent = match.get("start").get(0);
    IN endEvent = match.get("end").get(0);
    out.collect(OUT(startEvent, endEvent));
  }
}

```

- 处理超时的部分匹配

- 当一个模式上通过 `within` 加上窗口长度后，部分匹配的事件序列就可能因为超过窗口长度而被丢弃。可以使用 `TimedOutPartialMatchHandler` 接口来处理超时的部分匹配。这个接口可以和其他的混合使用。也就是说你可以在自己的 `PatternProcessFunction` 里另外实现这个接口。`TimedOutPartialMatchHandler` 提供了另外的 `processTimedOutMatch` 方法，这个方法对每个超时的部分匹配都会调用。

```

○ class MyPatternProcessFunction<IN, OUT> extends
  PatternProcessFunction<IN, OUT> implements
  TimedOutPartialMatchHandler<IN> {
    @Override
    public void processMatch(Map<String, List<IN>> match, Context ctx,
    Collector<OUT> out) throws Exception;
    ...
}

@Override
public void processTimedOutMatch(Map<String, List<IN>> match,
Context ctx) throws Exception;
  IN startEvent = match.get("start").get(0);
  ctx.output(outputTag, T(startEvent));
}
}

```

## 5.4. 时间处理

- 在 CEP 中，事件的处理顺序很重要。在使用事件时间时，为了保证事件按照正确的顺序被处理，一个事件到来后会先被放到一个缓冲区中，在缓冲区里事件都按照时间戳从小到大排序，当水位线到达后，缓冲区中所有小于水位线的事件被处理。这意味着水位线之间的数据都按照时间戳被顺序处理。
- 为了保证跨水位线的事件按照事件时间处理，Flink CEP 库假定水位线一定是正确的，并且把时间戳小于最新水位线的事件看作是晚到的。晚到的事件不会被处理。

```

○ PatternStream<Event> patternStream = CEP.pattern(input, pattern);

OutputTag<String> lateDataOutputTag = new OutputTag<String>("late-
data"){};

singleOutputStreamOperator<ComplexEvent> result = patternStream
  .sideOutputLateData(lateDataOutputTag)
  .select(
    new PatternSelectFunction<Event, ComplexEvent>() {...}
  );
}

DataStream<String> lateData = result.getSideOutput(lateDataOutputTag);

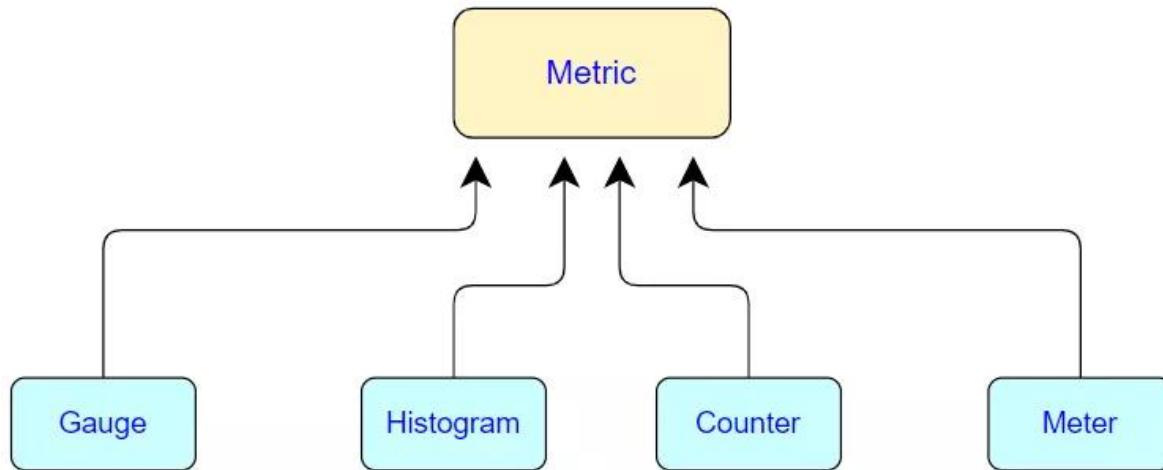
```

## 6. Flink Metrics

- metric 英['metrik] 美['metrik]
- Flink Metrics 是 Flink 集群运行中的各项指标，包含机器系统指标，比如：CPU、内存、线程、JVM、网络、IO、GC 以及任务运行组件(JM、TM、Slot、作业、算子)等相关指标。

- 通过flink metrics这些指标都可以获取到，避免任务的运行处于黑盒状态，通过分析这些指标，可以更好的调整任务的资源、定位遇到的问题、对任务进行监控。
- Flink Metrics 包含两大作用：
  - 实时采集监控数据。在 Flink 的 UI 界面上，用户可以看到自己提交的任务状态、时延、监控信息等等。
  - 对外提供数据收集接口。用户可以将整个 Flink 集群的监控数据主动上报至第三方监控系统，如：prometheus、grafana 等。

## 6.1. Metrics类别



- Flink一共提供了四种监控指标：分别为 Counter、Gauge、Histogram、Meter。
  - Gauge —— 最简单的度量指标，只是简单的返回一个值，比如当前实时读取kafka数据的条数
  - Counter —— 计数器，在一些情况下，会比Gauge高效，比如通过一个AtomicLong变量来统计一个队列的长度；
  - Meter —— 吞吐量的度量，也就是一系列事件发生的速率，例如TPS；
  - Histogram —— 度量值的统计结果，如最大值、最小值、平均值，以及分布情况等。

### 6.1.1. Count 计数器

- 统计指标的总量。写过 MapReduce 的开发人员就应该很熟悉 Counter，其实含义都是一样的，就是对一个计数器进行累加，即对于多条数据和多兆数据一直往上加的过程。其中 Flink 算子的接收记录总数 (numRecordsIn) 和发送记录总数 (numRecordsOut) 属于 Counter 类型。
- 使用方式：可以通过调用 counter(String name)来创建和注册 MetricGroup

```

public class MyMapper extends RichMapFunction<String, String> {
    private transient Counter counter;

    @Override
    public void open(Configuration config) {
        this.counter = getRuntimeContext()
            .getMetricGroup()
            .counter("myCustomCounter", new CustomCounter());
    }

    @Override
    public String map(String value) throws Exception {
        this.counter.inc();
        return value;
    }
}
  
```

### 6.1.2. Gauge 指标瞬时值

- Gauge是最简单的Metrics，它反映一个指标的瞬时值。比如要看现在TaskManager的JVM heap内存用了多少，就可以每次实时的暴露一个Gauge，Gauge当前的值就是heap使用的量。
- 使用前首先创建一个实现org.apache.flink.metrics.Gauge接口的类。返回值的类型没有限制。您可以通过在MetricGroup上调用gauge。

```
• public class MyMapper extends RichMapFunction<String, String> {  
    private transient int valueToExpose = 0;  
  
    @Override  
    public void open(Configuration config) {  
        getRuntimeContext()  
            .getMetricGroup()  
            .gauge("MyGauge", new Gauge<Integer>() {  
                @Override  
                public Integer getValue() {  
                    return valueToExpose;  
                }  
            });  
    }  
  
    @Override  
    public String map(String value) throws Exception {  
        valueToExpose++;  
        return value;  
    }  
}
```

### 6.1.3. Meter 平均值

- 用来记录一个指标在某个时间段内的平均值。Flink中的指标有Task算子中的numRecordsInPerSecond,记录此Task或者算子每秒接收的记录数。
- 使用方式：通过markEvent()方法注册事件的发生。通过markEvent(long n)方法注册同时发生的多个事件。

```
• public class MyMapper extends RichMapFunction<Long, Long> {  
    private transient Meter meter;  
  
    @Override  
    public void open(Configuration config) {  
        this.meter = getRuntimeContext()  
            .getMetricGroup()  
            .meter("myMeter", new MyMeter());  
    }  
  
    @Override  
    public Long map(Long value) throws Exception {  
        this.meter.markEvent();  
        return value;  
    }  
}
```

- ```

public class MyMeter implements Meter {
    int cnt = 0;
    int sum = 0;

    @Override
    public void markEvent() {
        sum += RandomUtils.nextInt(100, 200);
        cnt++;
    }

    @Override
    public void markEvent(long l) {
        sum += l;
        cnt++;
    }

    @Override
    public double getRate() {
        return sum / cnt;
    }

    @Override
    public long getCount() {
        return cnt;
    }
}

```

#### 6.1.4. Histogram 直方图

- Histogram 用于统计一些数据的分布，比如说 Quantile、Mean、StdDev、Max、Min 等，其中最重要一个是统计算子的延迟。此项指标会记录数据处理的延迟信息，对任务监控起到很重要的作用。
- 使用方式：通过调用 histogram(String name, Histogram histogram) 来注册一个 MetricGroup。

- ```

public class MyMapper extends RichMapFunction<Long, Long> {
    private transient Histogram histogram;

    @Override
    public void open(Configuration config) {
        this.histogram = getRuntimeContext()
            .getMetricGroup()
            .histogram("myHistogram", new MyHistogram());
    }

    @Override
    public Long map(Long value) throws Exception {
        this.histogram.update(value);
        return value;
    }
}

```

## 6.2. Metric Scope

- 每个 Metric 都会分配一个标识符和一组键值对，用来报告 Metric。
- Flink 的指标体系按树形结构划分，域相当于树上的顶点分支，表示指标大的分类。每个指标都会分配一个标识符，该标识符将基于 3 个组件进行汇报：
  - 注册指标时用户提供的名称；
  - 可选的用户自定义域；
  - 系统提供的域。
- 例如：
  - 如果 A.B 是系统域，C.D 是用户域，E 是名称，那么指标的标识符将是 A.B.C.D.E.
  - 可以通过设置 conf/flink-conf.yaml 里面的 metrics.scope.delimiter 参数来配置标识符的分隔符(默认“.”)。

### 6.2.1. User Scope

- 定义 User Scope 的方法：
  - 调用 MetricGroup#addGroup(String name)、MetricGroup#addGroup(int name)、MetricGroup#addGroup(String key, String value)。
  - 这些方法会影响 MetricGroup#getMetricIdentifier 和 MetricGroup#getScopeComponents 的返回值。

```
// 创建 Metric 时指定 Scope
counter = getRuntimeContext()
    .getMetricGroup()
    .addGroup("MyMetrics")
    .counter("myCounter");

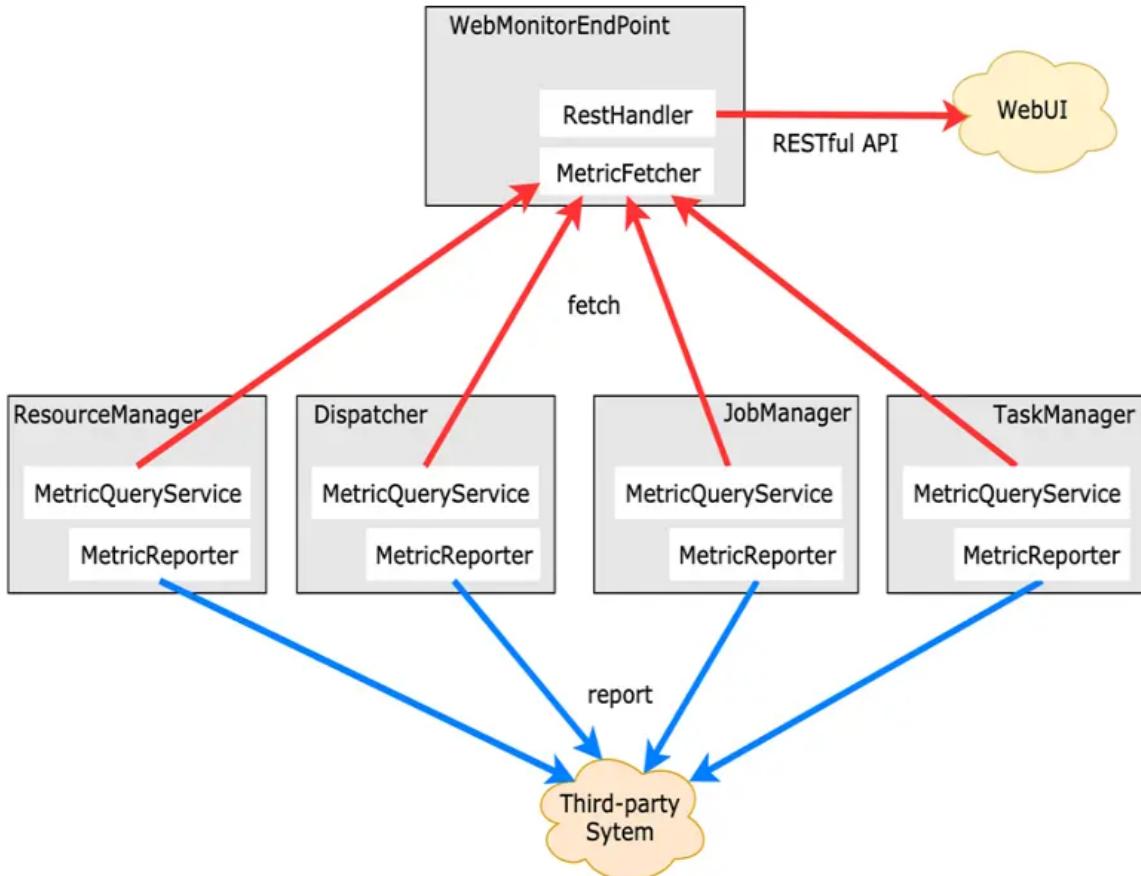
counter = getRuntimeContext()
    .getMetricGroup()
    .addGroup("MyMetricsKey", "MyMetricsValue")
    .counter("myCounter");
```

### 6.2.2. System Scope

- System Scope 包含 Metric 的上下文信息，例如注册在哪个 Task (<task\_name>) 或属于哪个 Job (<job\_name>)。
- 上下文信息可以通过 conf/flink-conf.yaml 配置。
  - metrics.scope.jm
    - 默认值: <host>.jobmanager
    - JobManager 的所有 Metric
  - metrics.scope.jm.job
    - 默认值: <host>.jobmanager.<job\_name>
    - JobManager 和 Job 的所有 Metric
  - metrics.scope.tm
    - 默认值: <host>.taskmanager.<tm\_id>
    - TaskManager 的所有 Metric
  - metrics.scope.tm.job
    - 默认值: <host>.taskmanager.<tm\_id><job\_name>
    - TaskManager 和 Job 的所有 Metric
  - metrics.scope.task
    - 默认值: <host>.taskmanager.<tm\_id><job\_name><task\_name><subtask\_index>

- Task 的所有 Metric
  - metrics.scope.operator
    - 默认值: <host>.taskmanager.<tm\_id><job\_name><operator\_name><subtask\_index>
    - Operator 的所有 Metric
- <host> | <job\_name> | <tm\_id> | <task\_name> | <operator\_name> | <subtask\_index> 可以作为变量使用。变量的数量或顺序没有限制，区分大小写。
- 例如：Operator Metric 的默认 Scope 格式为 <host>.taskmanager.<tm\_id><job\_name><operator\_name><subtask\_index>，生成的标识符类似 localhost.taskmanager.1234.MyJob.MyOperator.0.MyMetric 的形式；如果希望包含 Task 名称，并且忽略 TaskManager 信息，可以设置 metrics.scope.operator: <host>.<job\_name>.<task\_name>.<operator\_name>.<subtask\_index>，生成的标识符会变成 localhost.MyJob.MySource->\_MyOperator.MyOperator.0.MyMetric。
- 建议添加带有 ID 的变量（如：<job\_id>）保证唯一性，避免出现命名冲突的问题。所有可以使用的变量：
  - JobManager: <host>
  - TaskManager: <host>, <tm\_id>
  - Job: <job\_id>, <job\_name>
  - Task: <task\_id>, <task\_name>, <task\_attempt\_id>, <task\_attempt\_num>, <subtask\_index>
  - Operator: <operator\_id>, <operator\_name>, <subtask\_index>

### 6.3. Reporter



- Flink 允许向外部系统报告 Metric。

- 通过在 `conf/flink-conf.yaml` 中配置一个或多个 Reporter，可以将 Metric 暴露给外部系统。这些 Reporter 在启动时实例化。
  - `metrics.reporter.<name>.<config>`: Reporter 名称
  - `metrics.reporter.<name>.class`: Reporter 实现类
  - `metrics.reporter.<name>.factory.class`: Reporter 工厂类
  - `metrics.reporter.<name>.interval`: Reporter 调用间隔
  - `metrics.reporter.<name>.scope.delimiter`: Scope 标识符的分隔符（默认使用 `metrics.scope.delimiter`）
  - `metrics.reporter.<name>.scope.variables.excludes`: 可选项，以 ";" 分隔的变量列表，可以忽略这些变量
  - `metrics.reporters`: 可选项，以 "," 分隔的 Reporter 名称列表，表示应用哪些 Reporter，默认会包含所有配置的 Reporter。
- Reporter 必须至少配置 `class` 或 `factory.class` 属性（使用哪个取决于 Reporter 的实现）。
  - 配置示例

```

metrics.reporters: my_jmx_reporter,my_other_reporter

metrics.reporter.my_jmx_reporter.factory.class:
org.apache.flink.metrics.jmx.JMXReporterFactory
metrics.reporter.my_jmx_reporter.port: 9020-9040
metrics.reporter.my_jmx_reporter.scope.variables.excludes:job_id;task_attempt_num

metrics.reporter.my_other_reporter.class:
org.apache.flink.metrics.graphite.GraphiteReporter
metrics.reporter.my_other_reporter.host: 192.168.1.1
metrics.reporter.my_other_reporter.port: 10000

```

- 自定义 Reporter:
  - 实现 `org.apache.flink.metrics.reporter.MetricReporter` 接口
  - 如果要定时发送报告，实现 `Scheduled` 接口

## 7. Flink Backpressure

### back pressure

英 [bæk 'preʃə(r)] ⓘ ⓘ 美 [bæk 'preʃər] ⓘ ⓘ

背压；反压；前级（出口）压强

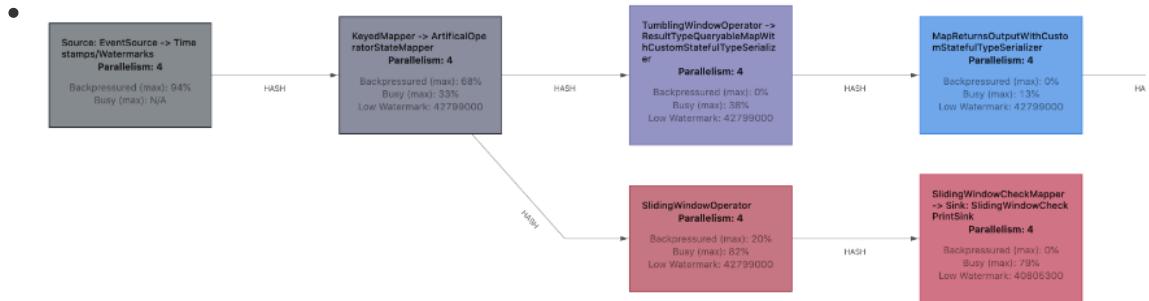
## 监控反压

Flink Web 界面提供了一个选项卡来监控正在运行 jobs 的反压行为。

### 7.1. 监控反压

- 如果你看到一个 task 发生 **反压警告**（例如： `High`），意味着它生产数据的速率比下游 task 消费数据的速率要快。

- 在工作流中数据记录是从上游向下游流动的（例如：从 Source 到 Sink）。反压沿着相反的方向传播，沿着数据流向上传播。
- Task (SubTask) 的每个并行实例都可以用三个一组的指标评价：
  - `backPressureTimeMsPerSecond`, subtask 被反压的时间
  - `idleTimeMsPerSecond`, subtask 等待某类处理的时间
  - `busyTimeMsPerSecond`, subtask 实际工作时间 在任何时间点，这三个指标相加都约等于 1000ms。
- 这些指标每两秒更新一次，上报的值表示 subtask 在最近两秒被反压（或闲或忙）的平均时长。当你的工作负荷是变化的时需要尤其引起注意。比如，一个以恒定50%负载工作的 subtask 和另一个每秒钟在满负载和闲置切换的 subtask 的 `busyTimeMsPerSecond` 值相同，都是 500ms。
- 在内部，反压根据输出 buffers 的可用性来进行判断的。如果一个 task 没有可用的输出 buffers，那么这个 task 就被认定是在被反压。相反，如果有可用的输入，则可认定为闲置，



- 闲置的 tasks 为蓝色，完全被反压的 tasks 为黑色，完全繁忙的 tasks 被标记为红色。中间的所有值都表示为这三种颜色之间的过渡色。
- | Detail  | SubTasks | TaskManagers | Watermarks | Accumulators | BackPressure                                     | Metrics             | FlameGraph |
|---------|----------|--------------|------------|--------------|--------------------------------------------------|---------------------|------------|
|         |          |              |            |              | Measurement: 3s ago   Back Pressure Status: HIGH |                     |            |
| SubTask |          |              |            |              | Backpressured / Idle / Busy                      | Backpressure Status |            |
| 0       |          |              |            |              | 100% / 0% / 0%                                   | HIGH                |            |
| 1       |          |              |            |              | 100% / 0% / 0%                                   | HIGH                |            |
| 2       |          |              |            |              | 0% / 100% / 0%                                   | OK                  |            |
| 3       |          |              |            |              | 0% / 100% / 0%                                   | OK                  |            |
| 4       |          |              |            |              | 0% / 100% / 0%                                   | OK                  |            |
| 5       |          |              |            |              | 49% / 51% / 0%                                   | LOW                 |            |
| 6       |          |              |            |              | 97% / 3% / 0%                                    | HIGH                |            |
| 7       |          |              |            |              | 100% / 0% / 0%                                   | HIGH                |            |
| 8       |          |              |            |              | 100% / 0% / 0%                                   | HIGH                |            |
| 9       |          |              |            |              | 100% / 0% / 0%                                   | HIGH                |            |
| 10      |          |              |            |              | 100% / 0% / 0%                                   | HIGH                |            |

- 如果你看到 subtasks 的状态为 **OK** 表示没有反压。**HIGH** 表示这个 subtask 被反压。状态用如下定义：
  - OK**:  $0\% \leq \text{反压比例} \leq 10\%$
  - LOW**:  $10\% < \text{反压比例} \leq 50\%$
  - HIGH**:  $50\% < \text{反压比例} \leq 100\%$

## 7.2. 反压原因

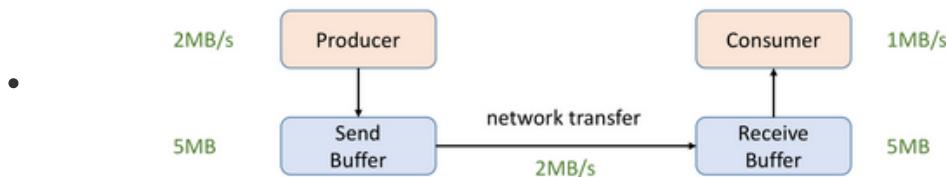
- 反压是流处理系统中用来保障应用可靠性的一个重要机制。由于流应用是7\*24小时运行，数据输入速率也不是一成不变，可能随时间产生波峰波谷，当某个处理单元由于到来的数据忽然增加，暂时性超出其处理能力时，就会出现数据在接收队列上累积，当数据的累积量超出处理单元的容量时，会出现数据丢失现象甚至因为系统资源耗尽而导致应用崩溃。为此，需要一种反压机制来告知

上游处理单元降低数据发送的速率，以缓解下游处理单元的压力。

- 流处理系统需要能优雅地处理反压 (backpressure) 问题。反压通常产生于这样的场景：短时负载高峰导致系统接收数据的速率远高于它处理数据的速率。
- 反压并不会直接影响作业的可用性，它表明作业处于亚健康的状态，有潜在的性能瓶颈并可能导致更大的数据处理延迟。
- 通常来说，对于一些对延迟要求不太高或者数据量比较小的应用来说，反压的影响可能并不明显，然而对于规模比较大的 Flink 作业来说反压可能会导致严重的问题。
- 反压会影响到两项指标：checkpoint 时长和 state 大小。
  - 前者是因为 checkpoint barrier 是不会越过普通数据的，数据处理被阻塞也会导致 checkpoint barrier 流经整个数据管道的时长变长，因而 checkpoint 总体时间 (End to End Duration) 变长。
  - 后者是因为为保证 EOS (Exactly-Once-Semantics, 准确一次)，对于有两个以上输入管道的 Operator，checkpoint barrier 需要对齐 (Alignment)，接收到较快的输入管道的 barrier 后，它后面数据会被缓存起来但不处理，直到较慢的输入管道的 barrier 也到达，这些被缓存的数据会被放到state 里面，导致 checkpoint 变大。

### 7.3. 网络监控

## 为什么需要网络流控

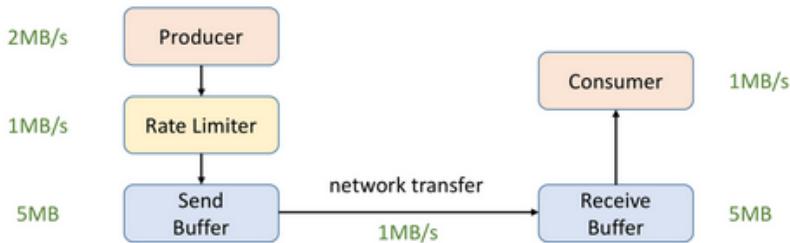


5秒钟后，将面临如下两种情况之一：

- ✓ bounded receive buffer: consumer 丢弃新到达的数据
- ✓ unbounded receive buffer: buffer 持续扩张，耗尽 consumer 内存

- 网络流控的图，Producer 的吞吐率是 2MB/s，Consumer 是 1MB/s，这个时候我们就会发现在网络通信的时候我们的 Producer 的速度是比 Consumer 要快的，有 1MB/s 的这样的速度差，假定我们两端都有一个 Buffer，Producer 端有一个发送用的 Send Buffer，Consumer 端有一个接收用的 Receive Buffer，在网络端的吞吐率是 2MB/s，过了 5s 后我们的 Receive Buffer 可能就撑不住了，这时候会面临两种情况：
  - 如果 Receive Buffer 是有界的，这时候新到达的数据就只能被丢弃掉了。
  - 如果 Receive Buffer 是无界的，Receive Buffer 会持续的扩张，最终会导致 Consumer 的内存耗尽。

## 网络流控的实现：静态限速

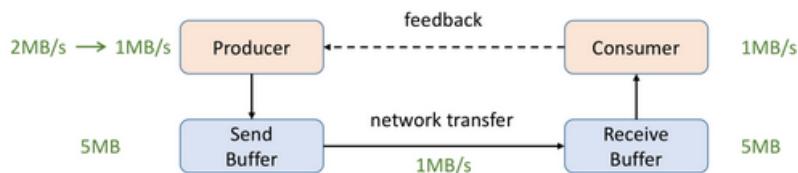


静态限流有两点限制：

- ✓ 通常无法事先预估 consumer 端能承受的最大速率
- ✓ consumer 承受能力通常会动态地波动

- 为了解决这个问题，我们就需要网络流控来解决上下游速度差的问题，传统的做法可以在 Producer 端实现一个类似 Rate Limiter 这样的静态限流，Producer 的发送速率是 2MB/s，但是经过限流这一层后，往 Send Buffer 去传数据的时候就会降到 1MB/s 了，这样的话 Producer 端的发送速率跟 Consumer 端的处理速率就可以匹配起来了，就不会导致上述问题。但是这个解决方案有两点限制：
  - 事先无法预估 Consumer 到底能承受多大的速率
  - Consumer 的承受能力通常会动态地波动
- 

## 网络流控的实现：动态反馈/自动反压



动态反馈分两种，广义上的反压机制都涵盖：

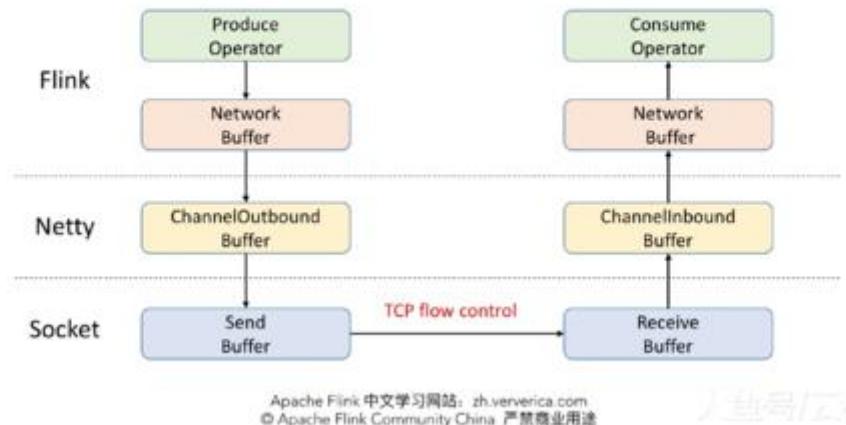
- ✓ 负反馈：接收速率小于发送速率时发生
- ✓ 正反馈：发送速率小于接收速率时发生

- 针对静态限速的问题我们就演进到了动态反馈（自动反压）的机制，我们需要 Consumer 能够及时的给 Producer 做一个 feedback，即告知 Producer 能够承受的速率是多少。动态反馈分为两种：
  - 负反馈：接受速率小于发送速率时发生，告知 Producer 降低发送速率
  - 正反馈：发送速率小于接收速率时发生，告知 Producer 可以把发送速率提上来
- flink 的反压又分为两个阶段，一个是 1.5 版本之前，一个是 1.5 版本以后

## 7.4. 反压流程

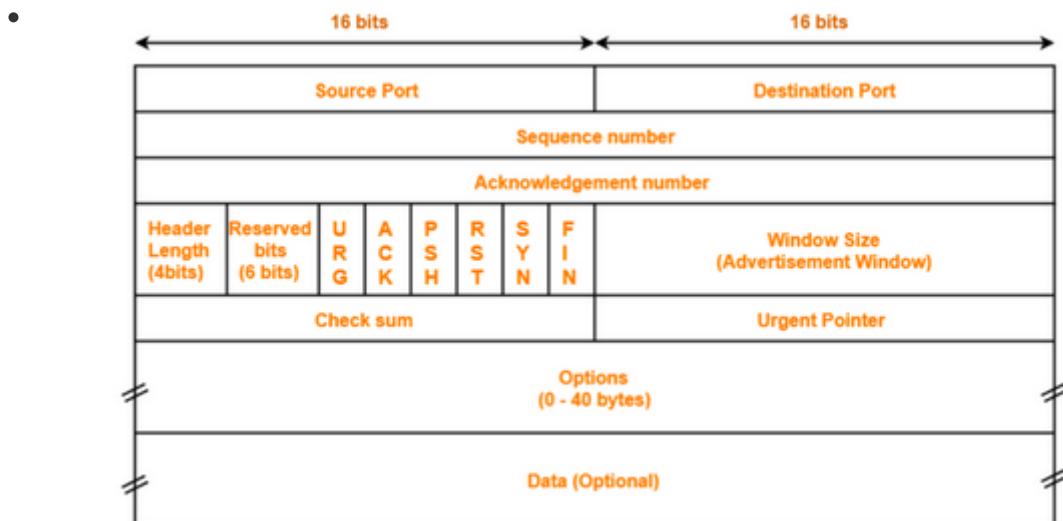
-

## Flink网络传输的数据流向

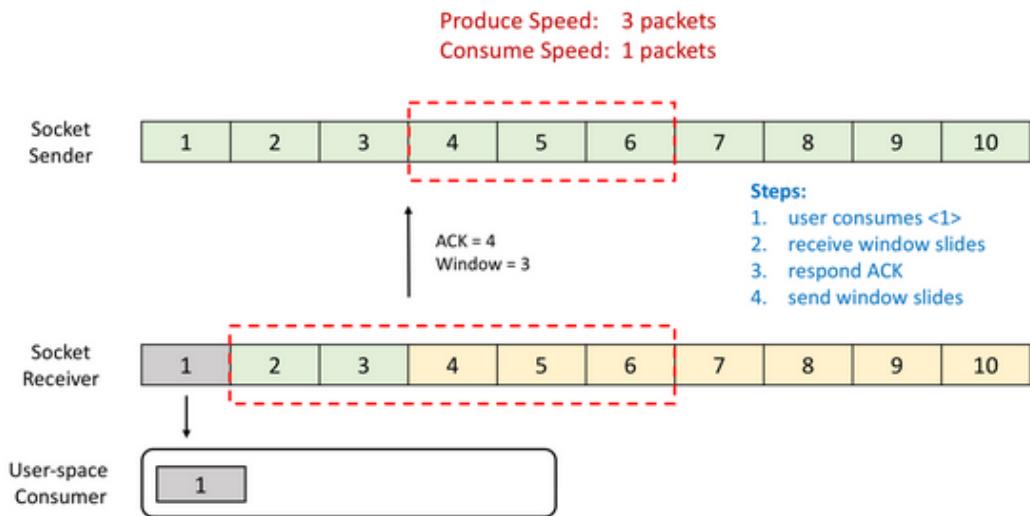


- Flink的反压是通过TCP的反压机制来控制的
- Flink 在做网络传输的时候基本的数据的流向，发送端在发送网络数据前要经历自己内部的一个流程，会有一个自己的 Network Buffer，在底层用 Netty 做出通信，Netty 这一层又有属于自己的 ChannelOutbound Buffer，因为最终是要通过 Socket 做网络请求的发送，所以在 Socket 也有自己的 Send Buffer，同样在接收端也有对应的三级 Buffer。学过计算机网络的时候我们应该了解到，TCP 是自带流量控制的。实际上 Flink (before V1.5) 就是通过 TCP 的流控机制来实现 feedback 的。
- 反压策略
  - before V1.5 使用 TCP-based 反压机制
  - since V1.5 使用 Credit-based 反压机制

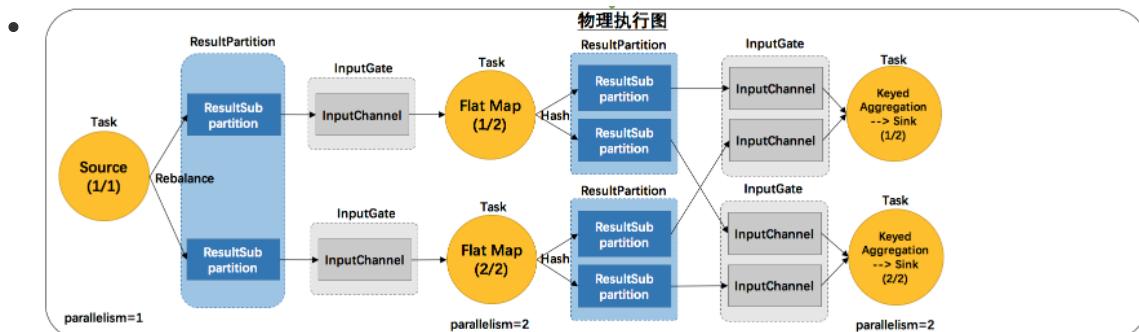
### 7.4.1. TCP流控机制



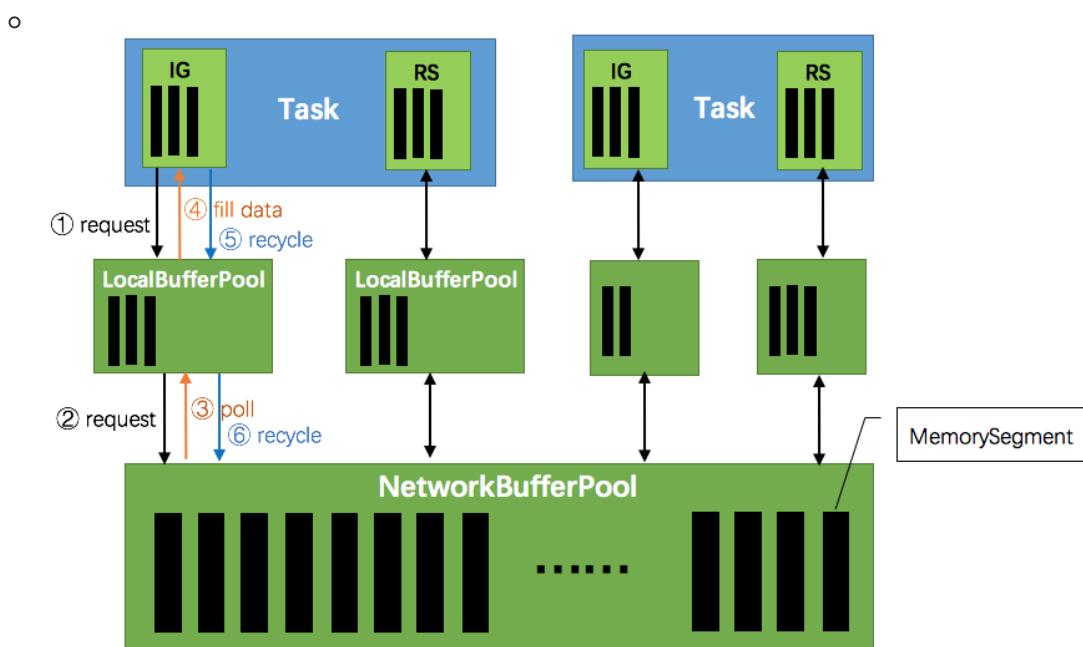
- TCP 包的格式结构。首先，他有 Sequence number 这样一个机制给每个数据包做一个编号，还有 ACK number 这样一个机制来确保 TCP 的数据传输是可靠的，除此之外还有一个很重要的部分就是 Window Size，接收端在回复消息的时候会通过 Window Size 告诉发送端还可以发送多少数据。
-



- TCP 当中有一个 ZeroWindowProbe 的机制，发送端会定期的发送 1 个字节的探测消息，这时候接收端就会把 window 的大小进行反馈。当接收端的消费恢复了之后，接收到探测消息就可以将 window 反馈给发送端从而恢复整个流程。TCP 就是通过这样一个滑动窗口的机制实现 feedback。



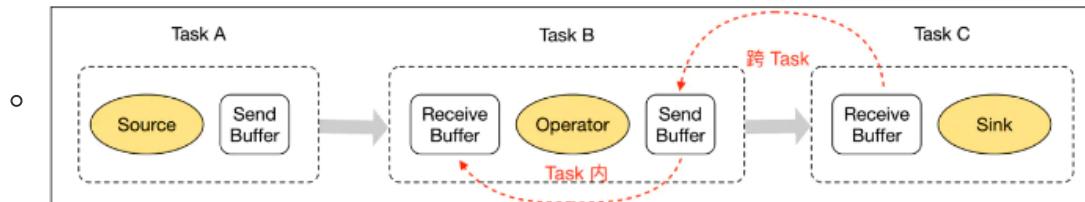
- 上游task向下游task传输数据的时候，有ResultPartition和InputGate两个组件。



- 每个task都会有自己对应的IG(inputgate)对接上游发送过来的数据和RS(resultPatation)对接往下游发送数据
- RP用来发送数据， IG用来接收数据
- 整个反压机制通过inputgate,resultPatation公用一个一定大小的memorySegmentPool来实现

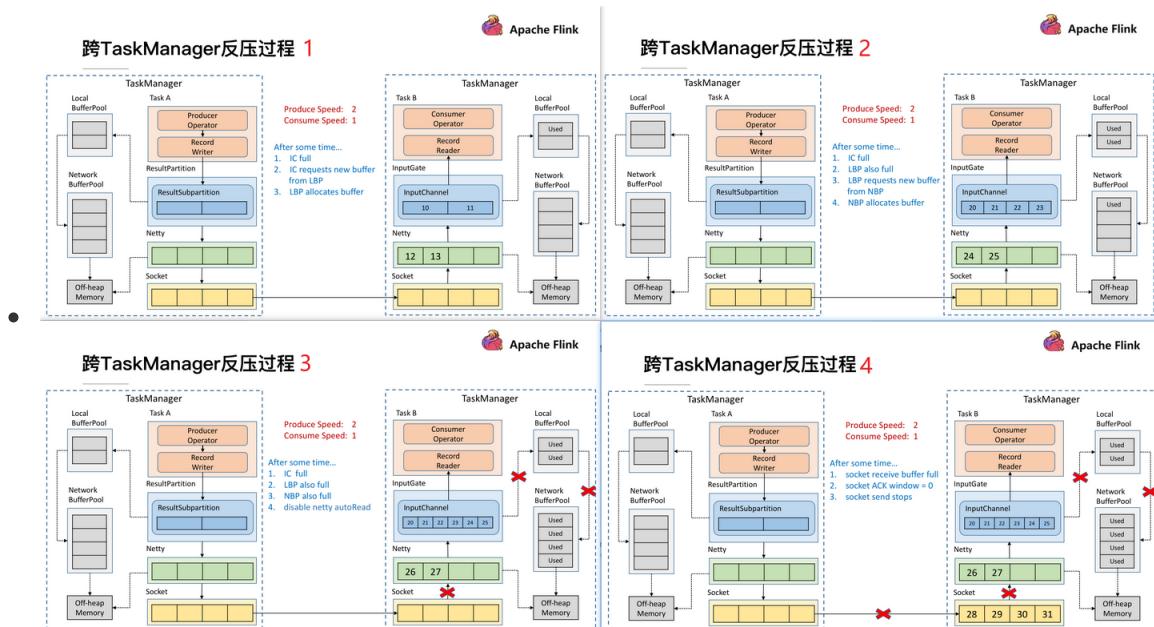
- 公用一个pool当接收上游数据时Decoder, 往下游发送数据时Encoder,都会向pool中请求内存memorySegment
- 因为是公共pool, 也就是说运行时, 当接受的数据占用的内存多了, 往下游发送的数据就少了
- 比如说你sink端堵塞了, 背压了写不进去, 那这个task的resultPatation无法发送数据了, 也就无法释放memorySegment了, 相应的用于接收数据的memorySegment就会越来越少, 直到接收数据端拿不到memorySegment了, 也就无法接收上游数据了, 既然这个task无法接收数据了, 自然引起这个task的上一个task数据发送端无法发送, 那上一个task又反压了
- 所以这个反压从发生反压的地方, 依次的往上游扩散直到source,这个就是flink的天然反压。

- 反压处理阶段划分



- 跨 TaskManager , 反压如何从 InputGate 传播到 ResultPartition
- TaskManager 内, 反压如何从 ResultPartition 传播到 InputGate

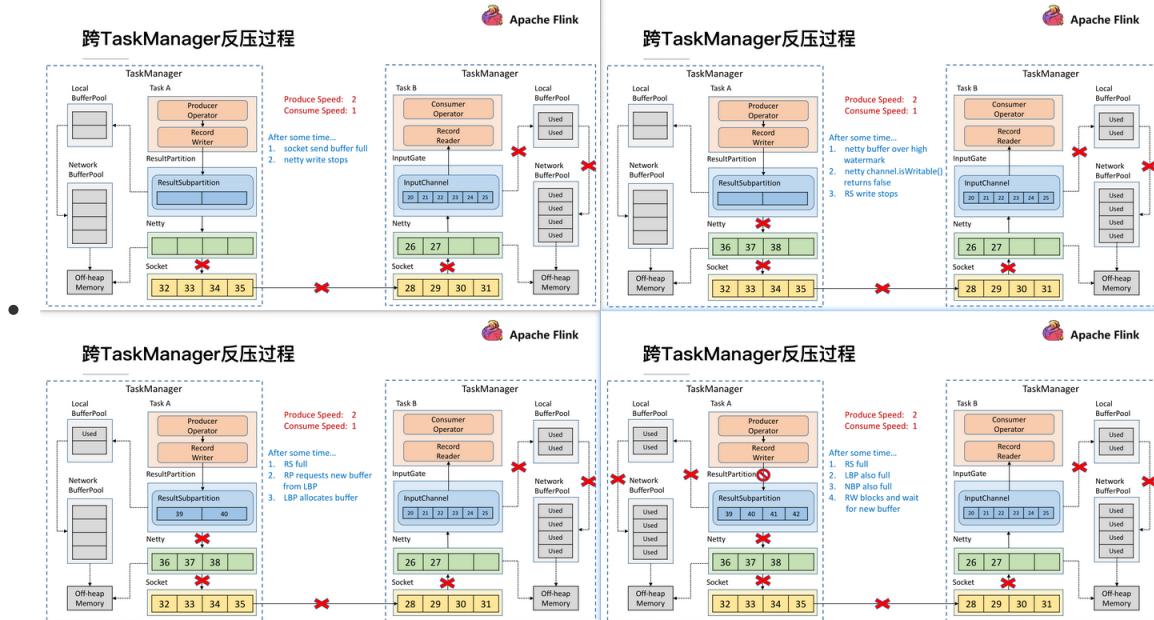
## 7.4.2. 跨TaskManager反压过程



- 对于一个 TaskManager 来说会有一个统一的 Network BufferPool 被所有的 Task 共享, 在初始化时会从 Off-heap Memory 中申请内存, 申请到内存的后续内存管理就是同步 Network BufferPool 来进行的, 不需要依赖 JVM GC 的机制去释放。有了 Network BufferPool 之后可以为每一个 ResultSubPartition 创建 Local BufferPool 。  
如上图左边的 TaskManager 的 Record Writer 写了 <1, 2> 这两个数据进来, 因为 ResultSubPartition 初始化的时候为空, 没有 Buffer 用来接收, 就会向 Local BufferPool 申请内存, 这时 Local BufferPool 也没有足够的内存于是将请求转到 Network BufferPool, 最终将申请到的 Buffer 按原链路返还给 ResultSubPartition, <1, 2> 这两个数据就可以被写入了。之后会将 ResultSubPartition 的 Buffer 拷贝到 Netty 的 Buffer 当中最终拷贝到 Socket 的 Buffer 将消息发送出去。然后接收端按照类似的机制去处理将消息消费掉。
- 因为速度不匹配就会导致一段时间后 InputChannel 的 Buffer 被用尽, 于是他会向 Local BufferPool 申请新的 Buffer , 这时候可以看到 Local BufferPool 中的一个 Buffer 就会被标记为

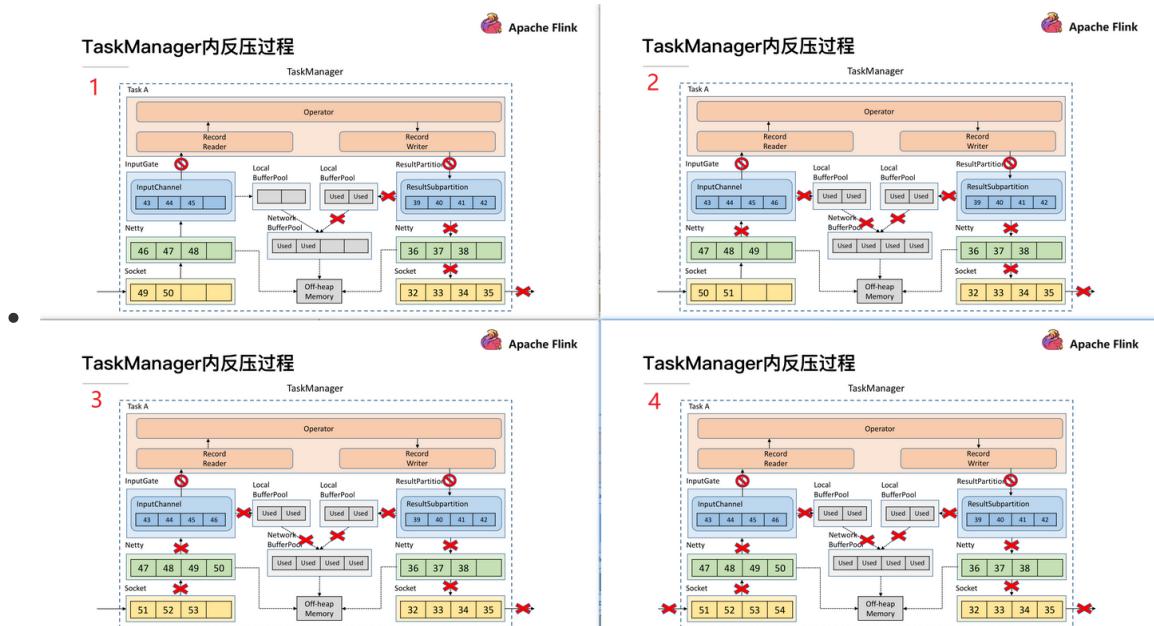
Used。

- 发送端还在持续以不匹配的速度发送数据，然后就会导致 InputChannel 向 Local BufferPool 申请 Buffer 的时候发现没有可用的 Buffer 了，这时候就只能向 Network BufferPool 去申请，当然每个 Local BufferPool 都有最大的可用的 Buffer，防止一个 Local BufferPool 把 Network BufferPool 耗尽。这时候看到 Network BufferPool 还是有可用的 Buffer 可以向其申请。
- 显然，再过不久 Socket 的 Buffer 也被用尽，这时就会将 Window = 0 发送给发送端（前文提到的 TCP 滑动窗口的机制）。这时发送端的 Socket 就会停止发送。



- 很快发送端的 Socket 的 Buffer 也被用尽，Netty 检测到 Socket 无法写了之后就会停止向 Socket 写数据。
- Netty 停止写了之后，所有的数据就会阻塞在 Netty 的 Buffer 当中了，但是 Netty 的 Buffer 是无界的，可以通过 Netty 的水位机制中的 high watermark 控制他的上界。当超过了 high watermark，Netty 就会将其 channel 置为不可写，ResultSubPartition 在写之前都会检测 Netty 是否可写，发现不可写就会停止向 Netty 写数据。
- 这时候所有的压力都来到了 ResultSubPartition，和接收端一样他会不断的向 Local BufferPool 和 Network BufferPool 申请内存。
- Local BufferPool 和 Network BufferPool 都用尽后整个 Operator 就会停止写数据，达到跨 TaskManager 的反压。

### 7.4.3. TaskManager内反压过程

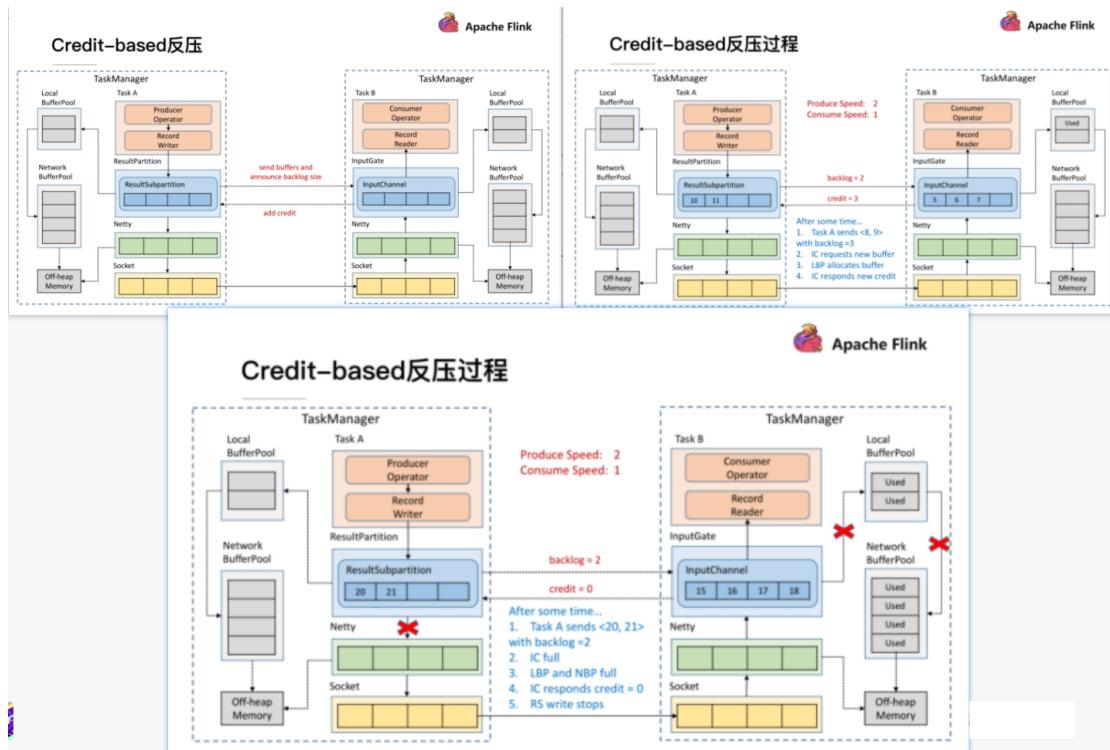


- 下游的 TaskManager 反压导致本 TaskManager 的 ResultSubPartition 无法继续写入数据，于是 Record Writer 的写也被阻塞住了，因为 Operator 需要有输入才能有计算后的输出，输入跟输出都是在同一线程执行，Record Writer 阻塞了，Record Reader 也停止从 InputChannel 读数据，这时上游的 TaskManager 还在不断地发送数据，最终将这个 TaskManager 的 Buffer 耗尽。

#### 7.4.4. Before 1.5 缺点

- 在一个 TaskManager 中可能要执行多个 Task，如果多个 Task 的数据最终都要传输到下游的同一个 TaskManager 就会复用同一个 Socket 进行传输，这个时候如果单个 Task 产生反压，就会导致复用的 Socket 阻塞，其余的 Task 也无法使用传输，checkpoint barrier 也无法发出导致下游执行 checkpoint 的延迟增大。
- 依赖最底层的 TCP 做流控，会导致反压传播路径太长，导致生效的延迟比较大。

#### 7.4.5. 1.5 After 革新



- 在 Flink 层面实现反压机制，就是每一次 ResultSubPartition 向 InputChannel 发送消息的时候都会发送一个 backlog size 告诉下游准备发送多少消息，下游就会去计算有多少的 Buffer 去接收消息，算完之后如果有充足的 Buffer 就会返还给上游一个 Credit 告知他可以发送消息
- 假设我们上下游的速度不匹配，上游发送速率为 2，下游接收速率为 1，可以看到图上在 ResultSubPartition 中累积了两条消息，10 和 11，backlog 就为 2，这时就会将发送的数据 <8,9> 和 backlog = 2 一同发送给下游。下游收到了之后就会去计算是否有 2 个 Buffer 去接收，可以看到 InputChannel 中已经不足了这时就会从 Local BufferPool 和 Network BufferPool 申请，好在这个时候 Buffer 还是可以申请到的。
- 过了一段时间后由于上游的发送速率要大于下游的接受速率，下游的 TaskManager 的 Buffer 已经到达了申请上限，这时候下游就会向上游返回 Credit = 0，ResultSubPartition 接收到之后就不会向 Netty 去传输数据，上游 TaskManager 的 Buffer 也很快耗尽，达到反压的效果，这样在 ResultSubPartition 层就能感知到反压，不用通过 Socket 和 Netty 一层层地向上反馈，降低了反压生效的延迟。同时也不会将 Socket 去阻塞，解决了由于一个 Task 反压导致 TaskManager 和 TaskManager 之间的 Socket 阻塞的问题。
- 基于 credit 的反压过程，效率比之前要高，因为只要下游 InputChannel 空间耗尽，就能通过 credit 让上游 ResultSubPartition 感知到，不需要在通过 netty 和 socket 层来一层一层的传递。另外，它还解决了由于一个 Task 反压导致 TaskManager 和 TaskManager 之间的 Socket 阻塞的问题。

### 7.5. 处理策略

### 7.5.1. 反压定位

- 定位造成反压问题的节点，通常有两种途径。
  - 反压监控面板；
  - Flink Task Metrics
- 前者简单易上手，适合简单分析，后者信息丰富，但需要更多背景知识，适合系统分析。
- 反压监控面板
  - Flink Web UI 的反压监控提供了 SubTask 级别的反压监控，通过周期性对 Task 线程的栈信息采样，得到线程被阻塞在请求 Buffer（意味着被下游队列阻塞）的比例来判断该节点是否处于反压状态。默认配置如下：
    - **OK:**  $0 \leq \text{Ratio} \leq 0.10$
    - **LOW:**  $0.10 < \text{Ratio} \leq 0.5$
    - **HIGH:**  $0.5 < \text{Ratio} \leq 1$
  - 值得注意的是，反压的根源节点并不一定会在反压监控面板体现出高反压，因为反压面板监控的是发送端，如果某个节点是性能瓶颈并不会导致它本身出现高反压，而是导致它的上游出现高反压。总体来看，如果我们找到第一个出现反压的节点，那么反压根源要么是就这个节点，要么是它紧接着的下游节点。
- Task Metrics
  - 参考前面的课程

### 7.5.2. 反压原因

- 系统资源
  - 首先，需要检查机器的资源使用情况，像CPU、网络、磁盘I/O等。如果一些资源负载过高，就可以进行下面的处理：
    - 尝试优化代码；
    - 针对特定资源对Flink进行调优；
    - 增加并发或者增加机器
- 垃圾回收
  - 性能问题常常源自过长的GC时长。这种情况下可以通过打印GC日志，或者使用一些内存/GC分析工具来定位问题。
- CPU/线程瓶颈
  - 有时候，如果一个或者一些线程造成CPU瓶颈，而此时，整个机器的CPU使用率还相对较低，这种CPU瓶颈不容易发现。比如，如果一个48核的CPU，有一个线程成为瓶颈，这时CPU的使用率只有2%。这种情况下可以考虑使用代码分析工具来定位热点线程。
- 线程争用
  - 跟上面CPU/线程瓶颈问题类似，一个子任务可能由于对共享资源的高线程争用成为瓶颈。同样的，CPU分析工具对于探查这类问题也很有用。
- 负载不均
  - 如果瓶颈是数据倾斜造成的，可以尝试删除倾斜数据，或者通过改变数据分区策略将造成数据的key值拆分，或者也可以进行本地聚合/预聚合。
  - 上面几项并不是全部场景。通常，解决数据处理过程中的瓶颈问题，进而消除反压，首先需要定位问题节点（瓶颈所在），然后找到原因，寻找原因，一般从检查资源过载开始。