Flink window API:

1. window的概念

1. 这里的窗口的概念和TCP中的流量控制的概念并不相同. 这里是使用窗口将无限的流切分为有限的流的一种方式, 用于进行一段时间内的统计 – 将流信息转化为统计信息. 这里的建模可以认为window是一个桶, 按照不同的条件实时地将数据放置到对应桶中
2. 窗口的实现: 通过将无限的数据流分发到有限大小的桶(bucket)中进行分析

2. window的类型

1. 根据窗口本身进行数据分流时使用的属性可以分为两大类, 然后依据窗口变化的方式可以分为多个种类:
    1. 时间窗口(通过数据到达的时间进行窗口地分配)
        1. 滚动时间窗口
        2. 滑动时间窗口
        3. 会话窗口
    2. 计数窗口(按照数据的个数进行截取)
        1. 滚动计数窗口
        2. 滑动计数窗口
2. 滚动窗口–`Tumbling Windows`(这里对于时间或者计数只是数据来源不同)
    1. 将用于区分的数据按照固定的长度连续切分为多个窗口.
    2. 每个窗口之间窗口长度相同, 窗口间没有重叠且窗口之间没有缝隙, 每个窗口是前闭后开的
    3. 对于一个数据而言其必定属于且只属于一个窗口
    4. 其参数只有一个`window size`
    5. `[StartPoint+k*WindowSize, StartPoint+k*WindowSize+WindowSize) (k∈Z+)`
3. 滑动窗口 – `Sliding Windows`:
    1. 是有固定窗口长度和滑动间隔构成. `[StartPoint+k*WindowSlide, StartPoint+k*WindowSlide+WindowSize) (k∈Z+)`
    2. **滚动窗口是`WindowSlide == WindowSize`的滑动窗口**
    3. 数据平均属于`WindowSize/WindowSlide`个窗口
4. 时间会话窗口 – `Session Windows`
    1. 需要设置一个timeout, 当一段事件没有收到新的事件的时候会关闭窗口 – 没有时间对齐, 窗口的长度不确定
    2. 由于timeout和时间是强绑定的, 所以没有基于计数的会话窗口

3. windowAPI的调用

1. 使用`window()`来定义一个窗口, 对于这个窗口可以做一些聚合以及其他操作. 调用的流程为`keyBy(分配窗口键值) -> window(以不同方式进行窗口的构建) -> reduce(对于窗口内的数据进行聚合生成窗口的最终结果)`
2. 以不同方式进行窗口构建的组件是窗口分配器(`window assigner`)
    1. 职责: 将每条输入的数据分发到正确的窗口中 (和窗口之间的关系是一对多)
    2. 窗口分类器的分类:
        1. 滚动窗口分类器 (tumbling window)
        2. 滑动窗口分类器 (sliding window)

    3. 会话窗口分类器 (session window)
    4. 全局窗口分类器 (global window) – 主要用于全局的计数工作
3. 对于窗口分配器分配出的数据进行操作: 窗口函数(window function)
    1. 职责: 定义了对于窗口中搜集到的数据做的计算操作
    2. 可以分为两类:
        1. 增量聚合函数: 通过类似RNN循环的方式进行处理, 会保留一个简单状态, 但是只在窗口结束后进行结果的输出
            1. (incremental aggregation function)
            2. 代表: `ReduceFunction`, `AggregateFunction`, 对于简单的滚动算子都是增量聚合
        2. 全窗口函数: 适用于一些需要窗口的全局知识才能处理的函数, 例如统计方差, 会等待窗口关闭然后进行运行
            1. 对于一些依赖完整数据集且提前计算复杂度较高用处较少的情况, 应使用全窗口函数
            2. 另外, 相比于增量聚合函数, 全窗口函数可以取得一些窗口的属性以及窗口的上下文, 在某些场景下适合使用
            3. (full window functions)
            4. 代表: `ProcessWindowFunction`, `WindowFunction`

4. 窗口API中的可选API

1. trigger: 触发器, 用于定义窗口什么时候发生关闭, 并输出窗口计算结果 (**注意: 由于需要考虑到数据的迟到问题, 窗口的关闭和数据的输出并不同时. 到达时间输出一次数据, 获取迟到数据输出补充数据, 当所有迟到数据被处理完成之后, 激活触发器关闭窗口**)
2. .evictor: 移出器, 用于将窗口中的某些数据进行移除, 删除一些不需要的数据(**例如用于移出测量的异常数据**)
3. .allowedLateness: 设置接口, 用于使窗口可以处理迟到的数据
    1. 例如由于到达处理窗口的数据处理顺序不同,导致到达时间并不按照数据的产生时间. 通过保留一段时间的窗口进行迟到数据的处理
4. sideOutputlateData: 使用侧数据流输出迟到的数据 – 这个接口是按照时间等待的一个补充, 如果在等待时间完成之后仍有迟到数据, 则将其放置到侧输出流中
5. getSideOutput: 获取侧输出流, 和上面的sideOutputlateData配套使用, 基于sideOutputlateData生成的流数据中剥离侧输出流

    注意: 这里关于latency的问题只在`even time window`中有效, 也就是在事件发生时间作为事件的时间标签. 而不是节点收到数据的时间

    注意: 这里所有的接口都是对于窗口操作的个性化, 所以其调用的顺序为`开窗 -> 窗口API -> 算子操作`

Flink的时间语义与watermark:

这一讲的主要目的是处理由于流程时间不一致导致的乱序以及超时问题

1. Flink中的时间语义
    1. Event Time: 事件创建时间, 一般通过事件的日志时间戳进行获取
    2. Ingestion Time: 数据进入Flink的时间, 如果事件数据中没有时间戳, 可以进行替代
    3. Processing Time: 执行当前操作算子的本地系统时间
2. 设置EventTime
    1. 不同的时间语义需要依据不同的应用需求进行设定, 在实际生产中一般更加关心事件的发生时间

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
// 传入事件时间:
dataStream.assignTimestampsAndWatermarks();
```

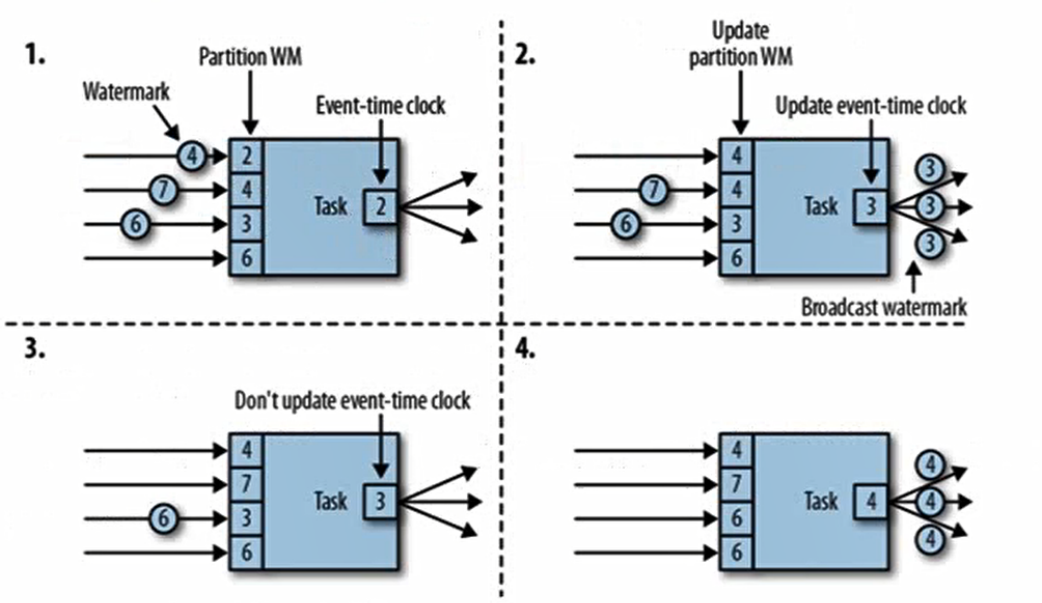
3. watermark

    这一部分没有太看懂

    1. 数据乱序: 如果以事件时间或者进入时间进行标记则可能发生数据乱序
    2. 原理: 当一个时间戳到达了窗口关闭的条件的时候并不立即关闭窗口, 而是等待一段时间之后再进行窗口的关闭
    3. 功能:
        1. 是一种衡量EventTime进展的机制, 可以设定延时触发
        2. 用于处理乱序时间, 其需要配合window进行使用
        3. 数据流中的watermark表示时间戳小于watermark的数据都已经到达了, 因此由window执行watermark是触发型的
        4. 可以用来让程序自己平衡延迟和结果的正确性
    4. 窗口关闭机制的时间顺序:

        `窗口正常范围内处理 -> watermark判断 -> windowLatency等待 -> 数据丢弃`

    5. 特点:
        1. watermark本身是一个特殊的数据记录, 其本身包含一个时间戳
        2. watermark随着数据流的方向一定是单调递增的
        3. watermark与数据的时间戳相关
        4. 插入一个watermark表示该数据流 中在watermark时间戳之前的数据都到齐了
        5. 当一个窗口发现接收到了一个watermark之后就会更新当前窗口的时间, 如果触发了窗口关闭的时间, 就进行窗口的关闭
4. watermark 的传递规则, 代码中的引入与设定:
    1. 传递方式 – 从上游到下游是广播传递 – 上游某个时间点之前的所有任务处理完成之后, 告知所有下游任务



1. 当一个节点有多个上游数据流的时候, 取上游watermark的最小值 — 也就是选取所有数据都获取完成的时间作为向下游广播的时间戳
2. 代码中关于watermark的引入:

```
        Time maxOutOfOrderness = Time.seconds(1);
        dataStream.assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor<SensorReading>
(maxOutOfOrderness) {
            @Override
            public long extractTimestamp(SensorReading sensorReading)
{
                return sensorReading.getTimestamp()*1000L;
            }
        });
```

1. 注意这里输入的时间戳是一个从Epoch(1970-01-01 00:00:00 UTC)开始的一个长整型毫秒数

5. 实际生产中watermark的使用:
    1. 一般使用watermark处理乱序数据
    2. 如果watermark的延迟时间给的过高会导致结果的生成延迟过慢
    3. 如果watermark的延迟设置的太小, 会导致输出得到不完整的结果
    4. 可以通过对于数据的延迟进行测试生成一个类似正态分布的曲线, 使用3σ的范围进行watermark的长度的设置
    5. 如果按照数据到来进行watermark的生成, 会在数据量过多的时候发生重复生成相同watermark的情况
    6. 如果使用定时watermark生成的方法, 则会在数据量过少的情况下平白生成watermark导致资源浪费
    7. 由于大数据一般用于处理稠密数据, 所以一般都是使用定时生成的生成方式
    8. 设置watermark延迟时间: `env.getConfig().setAutoWatermarkInterval(100);`
    9. window的offset可以用于处理不同时区之间的延迟, 用于在特定时间生成结果
    10. **注意: watermark的更新单元是一个输入源, 如果两组数据来自不同的数据源则其watermark并不会影响, 如果两个数据是通过串行的形式从一个数据源中进行获取, 则其watermark可能会互相影响, 需要自定义watermark生成器进行处理**

Flink状态管理:

1. Flink中的状态
    1. 状态是什么:
        1. 由一个任务维护, 并且用来计算某个结果的所有数据, 被称作该任务的状态
        2. 可以理解为一个本地变量, 用于进行任务的逻辑访问
        3. Flink的状态管理, 指的是对于状态一致性, 故障处理, 以及高效存储与访问相关的功能
        4. 状态和任务直接相关, 只能在同一个任务算子进行关联
    2. 状态的分类:
        1. managed state: 由Flink管理了的状态
        2. raw state: 需要个人声明以及实现
2. 算子状态(Operator State)
    1. 算子状态的作用范围是限定为算子任务
    2. 只要是同一个任务在同一个分区中运行, 则其是共享一个算子状态的
    3. 算子状态的数据结构:
        1. 列表状态: 表现为一组数据列表, 在并行度重新分配时会平均分给所有拆分出的节点
        2. 联合列表状态: 也是一组列表, 在并行度重新分配时将所有的列表合并并分发到所有拆分出的节点
        3. 广播状态: 适用于在同一个算子上的多项任务间进行状态的共享, 也就是同一个任务中的所有节点的状态完全一样
3. 键控状态(分组状态)(Keyed State)
    1. 根据输入流中定义的键值来维护和访问
    2. 相比于算子状态更加普遍
    3. 输入键值不同的数据只能访问当前键值对应的状态数据
    4. 键控状态数据类型:
        1. 值状态(value state): 将状态表现为单个的值(由于隐式的基于键值构建了键值对, 所以状态可以只是一个值)
        2. 列表状态(List state): 将状态表现为一组数据的列表 – 一般意义是当一个对象有多个相同类型的值的时候
        3. 映射状态(Map State): 当状态由多个不同类型的状态值构成, 则其是一个映射状态
        4. 聚合状态(Reducing state & aggregation state): 将状态表现为一个用于聚合操作的列表
4. 状态后端(State Beckends)
    1. 状态的储存, 访问以及维护, 是通过一个**可插入的组件**进行决定的, 这个组件就**被叫做状态后端**
    2. 状态后端的主要任务: **本地的状态管理**, 以及**将检查点写入远程存储**
    3. Flink中提供的三个状态后端:
        1. MemoryStateBackend: 将状态存储于内存中, 将其存储在TaskManager的JVM的堆上(也就是本地存储), 将CheckPoint存储在JobManager的内存中. **快速, 低延迟, 不稳定**
        2. FsStateBackend: 将状态checkpoint存储于远程的持久化文件系统中, 对于本地状态也是存放于TaskManager的JVM堆上. **高效的本地访问, 容错性更强**
        3. RocksDBStateBackend: 将所有状态序列化之后存入本地的RocksDB中进行存储
            1. RocksDB是基于FaceBook的LabDB研发的NoSql数据库, 以键值对的形式进行保存, 支持内存和硬盘双状态的保存. **支持taskManager中数据过大的情况, 访问速度较慢, 可以灵活的使用硬盘空间, 稳定性更佳**
    4. 关于状态后端的配置项:

1. 在Flink的配置文件中, `Fault tolerance and checkpointing` 栏目, `state.backend` 项中. 默认设置为 `filesystem`. 可以设置为 `jobmanager`, `filesystem`, `rocksdb` 三种或者也可以自定义模块并输入对应Factory类的名称
2. 在配置文件中也可以设置对应的重启方式: `jobmanager.execution.failover-strategy : region` — 也就是当一个taskmanager挂掉之后只重启该taskmanager上的任务, 而不是进行全部重启