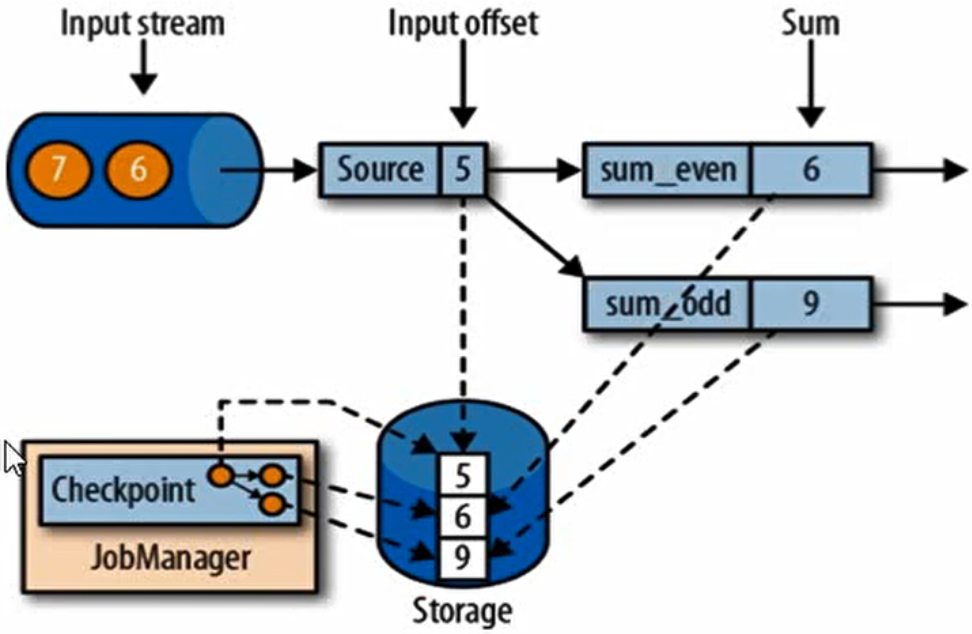
Flink ProcessFunction API (最底层API)

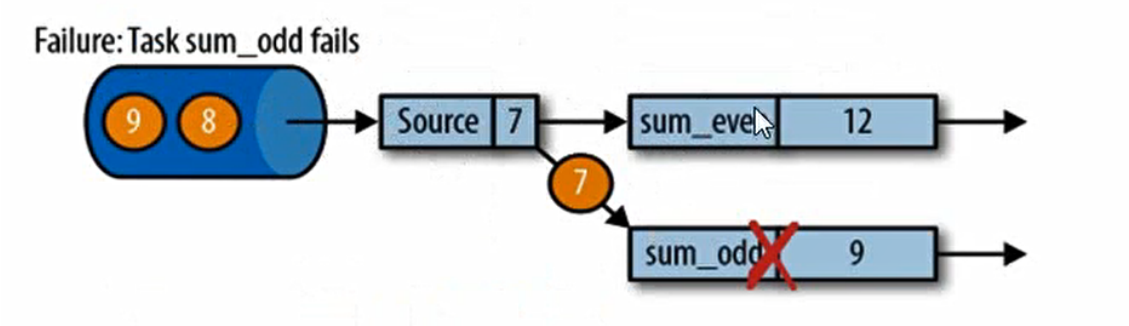
1. ProcessFunctionAPI 可以实现的功能:
    1. 访问时间戳,
    2. 访问watermark,
    3. 注册定时事件,
    4. 输出特殊事件等(例如超时事件)
2. ProcessFunction 常用的方面:
    1. 构建自定义的事件驱动应用
    2. 实现自定义的业务逻辑 (例如 FlinkSQL 是使用ProcessFunction进行实现的)
    3. 实现侧输出流的处理
3. 8个提供的processFunction, 通过流数据的process方法进行调用, 需要实现 `processElement`
    1. ProcessFunction:
        1. 一般流的processFunction
    2. KeyedProcessFunction:
        1. keyby之后的流的processFunction
    3. CoProcessFunction
        1. 基于connectedStream进行处理的function
    4. ProcessJoinFunction
        1. 基于Join后的流的processFunction
    5. BroadcastProcessFunction
        1. 广播流
    6. KeyedBriadcastProcessFunction\
        1. keyby之后进行广播
    7. ProcessWindowFunction
        1. window
    8. ProcessAllWindowFunction
        1. windowAll
4. ProcessFunction的一般接口:
    1. `OnTimer`: 需要提前定义一个计时器, 当计时器被触发的时候进行实现
    2. `processElement`: 需要实现的处理流程
    3. `extends AbstractRichFunction`: 所有richFunction中可以实现的功能在ProcessFunction中都可以实现
    4. `context.output`: 进行侧输出流的生成, 通过 `主流.getSideOutput` 进行获取

Flink的容错机制:

1. 一致性检查点
    1. 是flink的核心错误处理机制



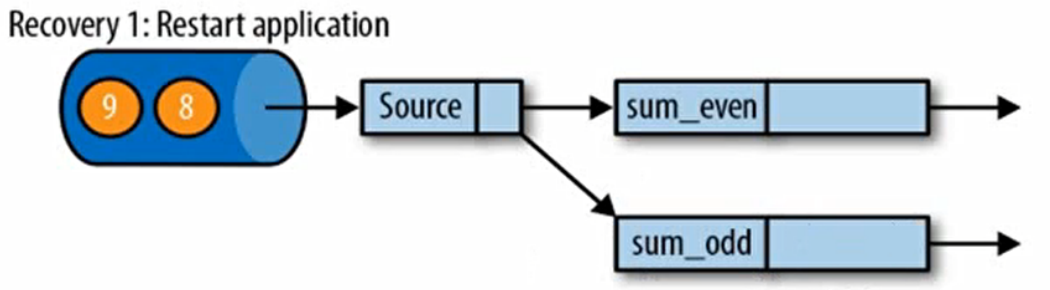
    2. `一致检查点`: 指的是在刚刚处理完某个输入数据的时候为所有的任务建立一份状态快照, 建立快照的时候每个节点的实际时间可能并不相同, 但是快照中不会未完成流程的数据的影响. 由于一般使用Kafka, 对于原始数据是可以重读的, 所以只进行状态的保存
    3. 在jobmanager中的checkpoint的保存是通过构建虚拟的内存网络, 然后为每个节点保存其对应的状态值
2. 从检查点恢复状态



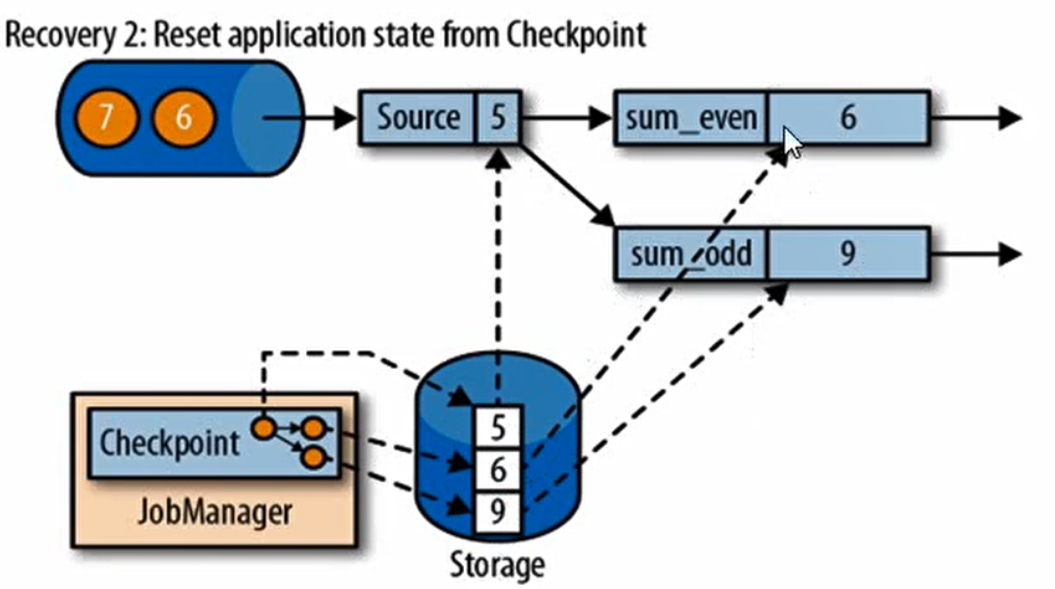
如果发生故障, Flink会使用最近的检查点来恢复应用的状态然后重新执行

执行步骤

1. 重启应用



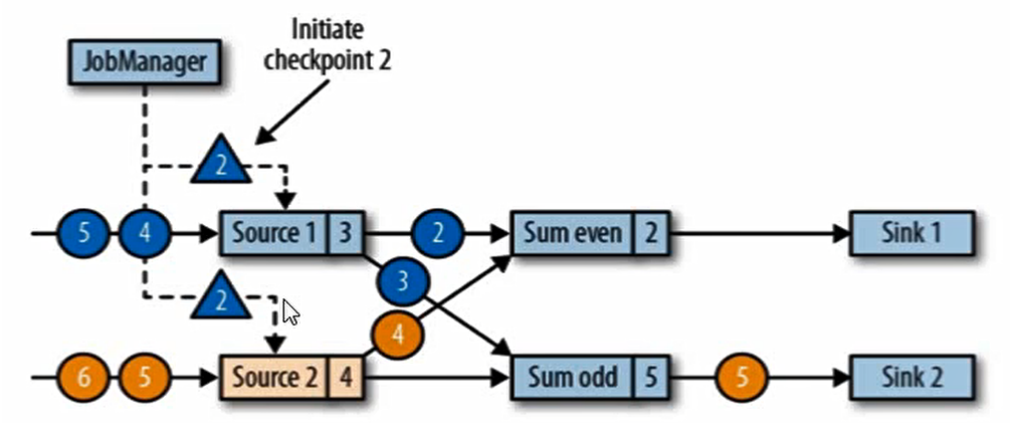
2. 从check point中读取数据恢复状态**(如果涉及到并行度的调整则会进行数据的拆分和合并)**



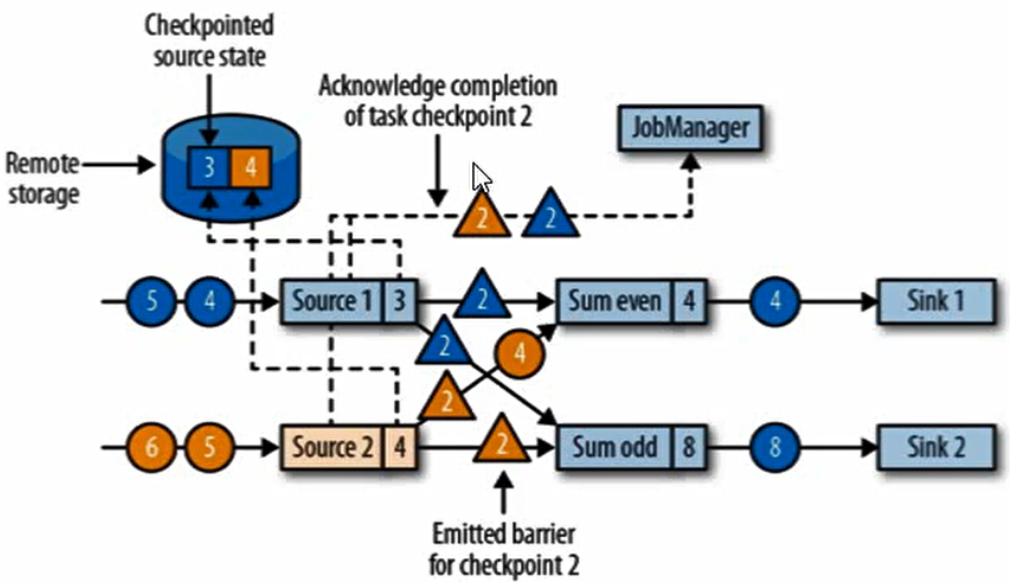
3. 对于kafka输入进行重置偏移量, 开始消费并处理发生故障之前的所有数据

   保证了`精确一致`的状态一致性(所有数据处理且只处理一次)

3. Flink检查点算法
   1. 简单想法: 暂停应用, 将状态保存到检查点, 然后恢复应用 — 阻塞式快照
   2. Flink改进实现:
      1. 基于Chandy-lamport算法的分布式快照 — 异步快照: 以节点为单位进行快照, 将状态组合为最终快照
         1. 在目标数据后续添加一个标记(checkpoint barrier), 所有在该标记之前的数据都会被包含到检查点中, 而之后的数据会被保存到后续的检查点中
      2. 将检查点的保存和数据处理分离开, 不进行全局的暂停
   3. 对于多流模型的检查点实现:
      1. 输入的时候有多个数据流
      2. jobmanager开始检查点2生成任务 – 向每个数据流中插入检查点通知信号:

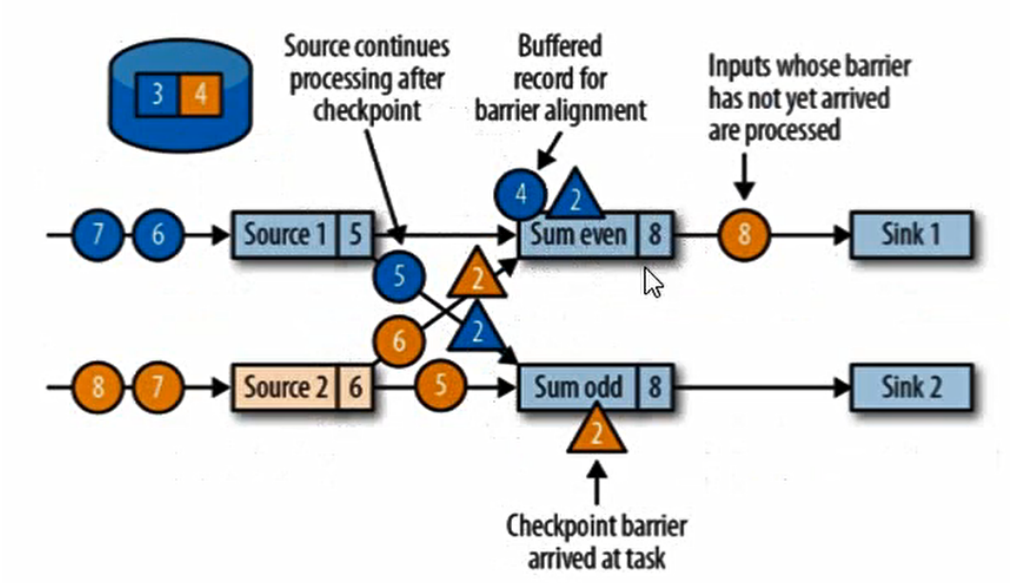


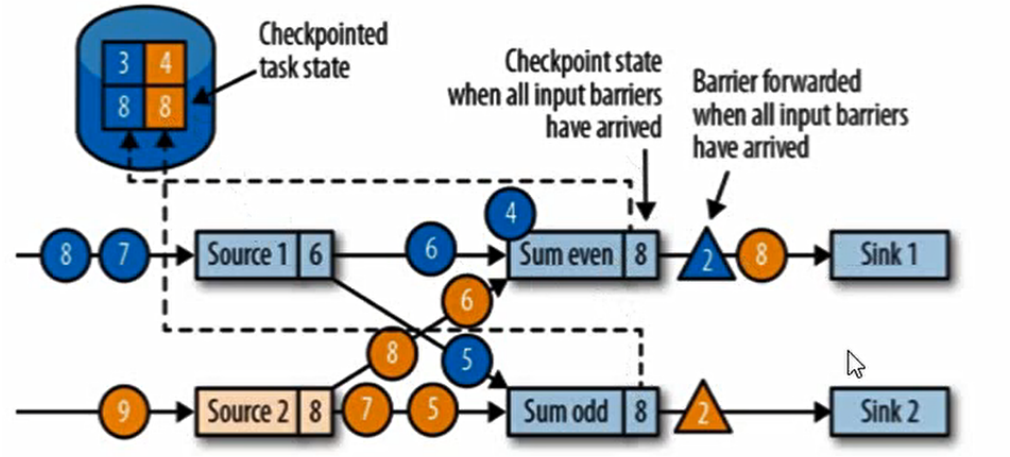
      3. 接收到通知的数据源节点进行快照保存并向JobManager发送回执, 并将检查点通知向下进行广播:



4. 当下游的节点确认上游的所有通知都被接收则进行快照, **如果在快照通知被完整收到之前接收到了快照通知后续的数据则对数据进行缓存, 暂不处理**

   这里将4号数据进行了缓存以等待黄色2号检查点的到来





5. 当JobManager收到所有快照任务全部被完成则确认检查点构建完成

4. 保存点(save point)

   1. 保存点是Flink中提供的一种自定义的镜像保存功能
   2. 保存点生成的算法和检查点是完全相同的, 所以可以认为保存点是拥有额外元数据的检查点, 可以通过Flink命令进行手动调用

    3. 保存点并不是Flink自动创建的, 而是需要用户或者外部程序进行手动的触发
    4. 保存点一般用于: 手动故障恢复, 有计划的手动备份, 应用更新, Flink版本迁移, 暂停与重启应用等
    5. 保存点进行应用更新的前提是节点的状态不能发生变化, 例如不涉及节点状态的小bug可以通过该方式进行实现. 因为保存点的数据实际上是节点状态的集合
5. 检查点的代码配置: 详见代码, 在开启检查点的设置之后还需要设置状态恢复的方式

Flink的状态一致性:

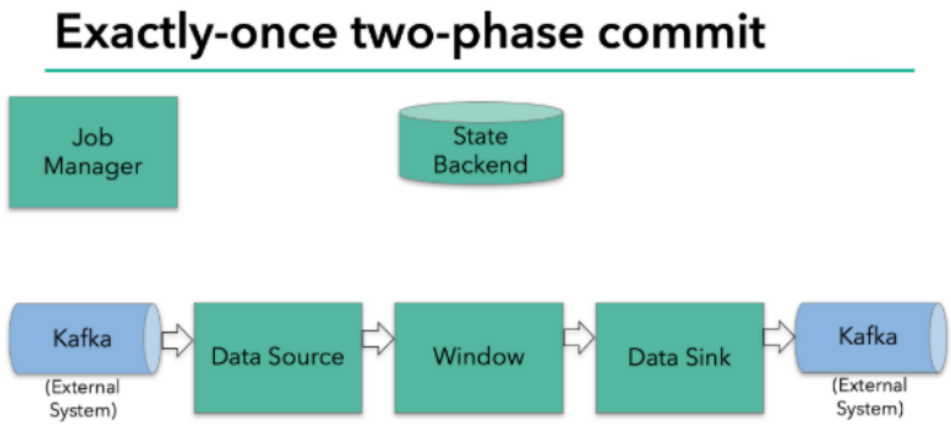
1. 状态一致性的概念:
    1. 这里的状态就是算子任务的状态
    2. 所谓一致性就是要保证计算结果的准确性(注意: 不是计算操作的一致性, 不是限定操作只能被执行一次)
    3. 表现为:
        1. 计算出的结果没有数据丢失也没有数据被重复计算
        2. 当遇到故障的时候可以进行恢复, 恢复完成后的计算结果也是完全正确的
2. 状态一致性的等级分类:
    1. at most once: 当发生任务故障的时候什么都不干, 直接进行重启, 丢失造成错误的数据, 对任何数据最多处理一次
    2. at least once: 当错误发生时不丢弃数据, 而是重新执行数据, 对每个任务可能被执行多次
    3. exactly once: 当错误发生时没有事件丢失, 内部状态仅更新一次
3. 一致性 – 检查点(checkpoint): 通过状态快照进行任务恢复, 检查点时错误恢复的核心
4. 端到端(end-to-end)状态一致性: 真实系统是一个复杂的系统, 不止要保证Flink内部的一致性, 还要保证外部组件的一致性
    1. 端到端的一致性, 取决于整个系统中一致性最弱的组件
    2. 端到端结构: `source -> flink -> sink`
        1. source端的一致性: 可重设数据的读取位置
        2. sink端一致性: 故障恢复时数据不会重复写入外部系统
            1. 幂等写入: (这里的命名借鉴了e^x的任意次倒数都是e^x的性质)
                1. 一个写入操作可以被重复执行多次, 但是只导致一次结果的更改, e.g. 以键值对的方式进行写入, 比如写入Redis
                2. 幂等写入的问题: 由于进行KV写入, 会导致中间结果的不一致
            2. 事务写入: 构建对应着checkpoint的写入事务, 只有当checkpoint完成的时候才输出到Sink系统
                1. 实现方式:
                    1. 预写日志
                        1. 先将结果数据当成状态进行保存, 当收到了checkpoint完成的通知的时候在一次性写入Sink系统
                        2. 简单易实现, 可以通过DataStreamApi – GenericWriteAheadSink模板类进行实现
                        3. **由于其写入还是批处理, 会影响处理的效率**
                    2. 两阶段提交
                        1. 当sink接收到一个checkpoint通知的时候, 提交之前的数据, 并开启一个新的事务
                        2. 在两个checkpoint之前, 只进行数据的预提交, 而不进行数据的写入
                        3. 支持的接口为: TwoPhaseCommitSinkFunction
                        4. **两阶段提交对于外部系统的要求(配置要求):**
                            1. 外部系统提供事务功能或者能够使用sink函数来模拟事务操作
                            2. 在checkpoint期间能够向事务中写入数据
                            3. 在收到checkpoint完成通知之前, 事务需要是等待提交状态, **不能发生超时 – 需要超时时间的匹配**

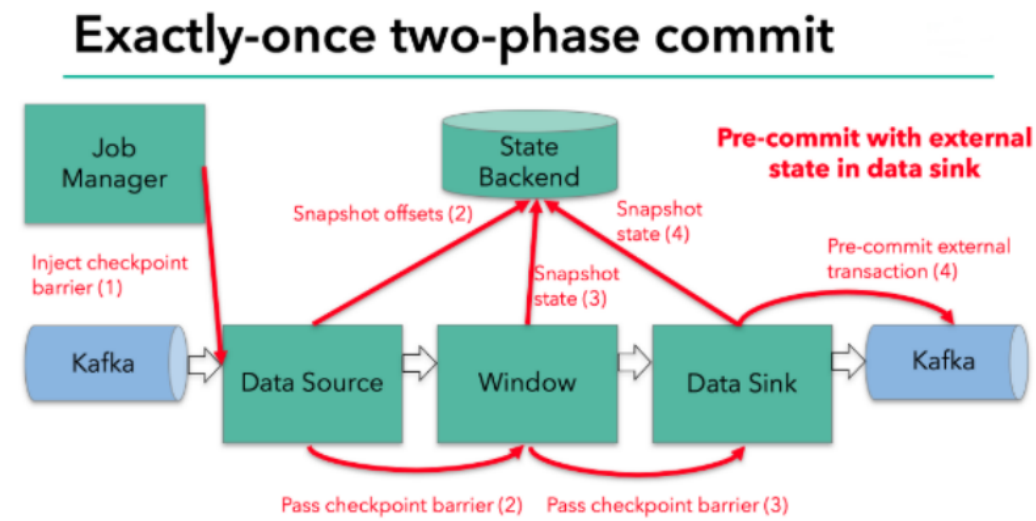
4. 当checkpoint生成失败, 事务需要被重置
5. 提交事务之后必须是幂等操作

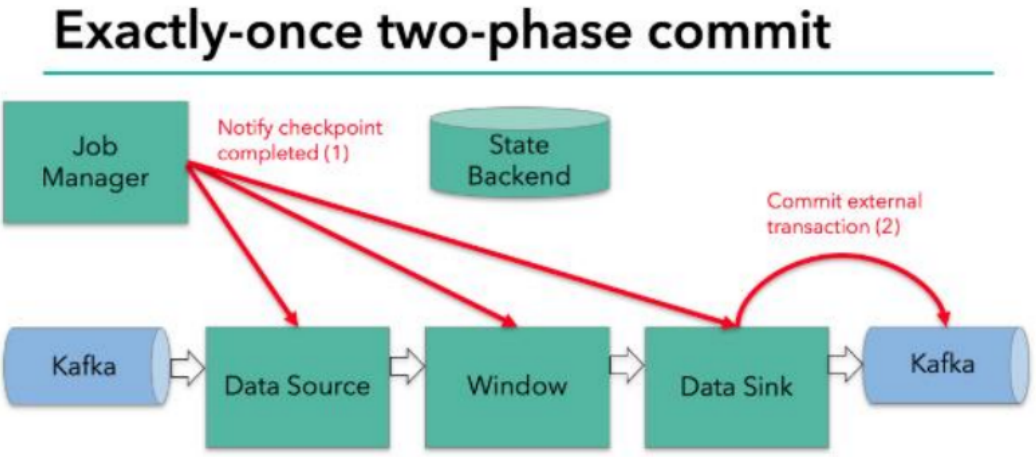
不同 Source 和 Sink 的一致性保证

sink \ source	不可重置	可重置
任意（Any）	At-most-once	At-least-once
幂等	At-most-once	Exactly-once （故障恢复时会出现暂时不一致）
预写日志（WAL）	At-most-once	At-least-once
两阶段提交（2PC）	At-most-once	Exactly-once

5. 端到端的精确一次(exactly-once)保证 (Flink和Kafka是最佳的流式处理配置)
    1. 内部使用checkpoint
    2. source使用Kafka保证可以进行重读
    3. sink使用Kafka producer机制作为sink, 实现两阶段提交
    4. 在`exactly-once`情况下的整体架构:







6. Flink+Kafka端到端状态一致性的保证
    1. 在kafka中开启事务, 正常写入kafka分区日志但是设置为未提交
    2. 当JobManager触发checkpoint操作, 发送checkpoint barrier
    3. 当sink收到JobManager回复的checkpoint完成通知则正式提交
    4. kafka关闭之前的事务