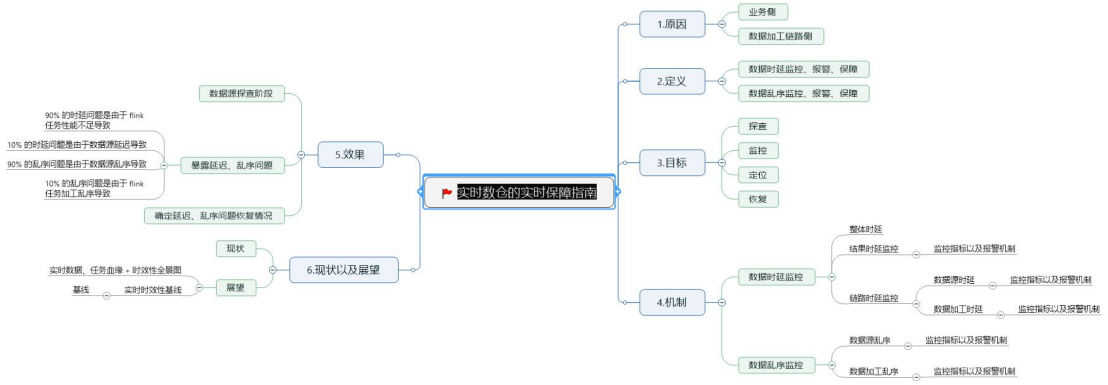
# 背景

所有的数据建设都是为了用户更快、更方便、更放心的使用数据。

在用户使用实时数据的过程中，最影响用户体感的指标有两个：

数据质量：实时数据产出的准确性。举个例子：实时数据在某些场景下不能保障端到端 exactly-once，因此实时与离线相同口径的数据会有 diff。而 1% 和 0.01% 的 diff 给用户的体验是完全不同的。

数据时效：实时数据产出的及时性。举个例子：延迟 1min 和 延迟 1ms 的用户体验也是完全不同的。




而本文主要对数据时效保障进行解读。

先说本文结论，通过以下两个指标就已经能监控和判定 90% 数据延迟、乱序问题了。

「**数据延迟监控**：flink 消费上游的 lag（比如看消费 kafka lag 情况）」

「**数据乱序监控**：Task/Operator numLateRecordsDropped 可以得到由于乱序导致窗口的丢数情况」

**数据时效保障就是对数据产出延迟、数据乱序的监控报警能力的构建、保障方案规范化的建设。**

# 数据时效包含哪些内容

实时数据时效保障可分为两部分：

**数据时延监控、报警、保障**：衡量实时数据产出的延迟情况，设定报警阈值，超过阈值触发报警。并且需要对数据产出延迟有一个全链路的视角，保障数据产出延迟在预期范围内；

**数据乱序监控、报警、保障**：乱序是实时任务处理中要关注的一个重要指标，如果数据源乱序非常严重的话，会影响窗口类任务产出的实时数据质量，所以我们也需要对齐进行监控以及保障。

**乱序的本质其实就是数据的延迟。乱序是一种特殊的延迟，数据延迟导致的一种结果。**

## 时效性监控以及保障的目标

探查：了解数据源的延迟、乱序情况。针对数据源的延迟、乱序情况可以针对性优化。也对此能提出合理的 SLA 保障；

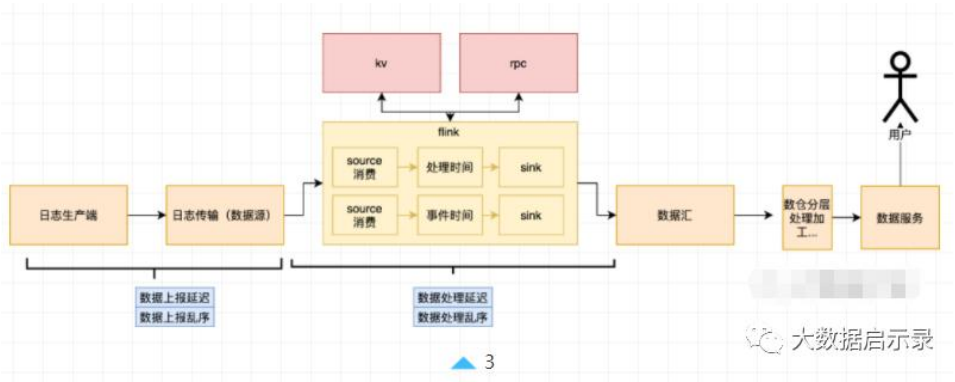
监控：针对具体延迟、乱序严重程度设定报警阈值，让开发可以快速感知问题；

定位：根据延迟、乱序报警快速定位数据延迟、乱序导致的质量问题；

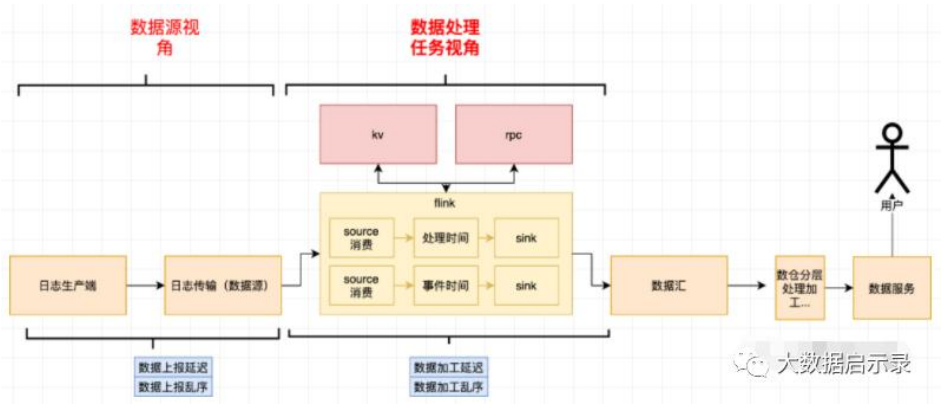
恢复：问题解决完成之后，可以根据监控查看到实际的效果；

## 怎么去做数据时效监控以及保障

接下来我们「**对症（延迟、乱序情况）**下药**（监控、报警、保障措施）**」，先分析在数据生产、传输、加工的过程中哪些环节会导致数据的延迟以及乱序。




通过分析上述数据生产、传输、加工链路之后，我们可以发现能从「**数据源、数据处理任务**」两个不同的维度去分析会导致延迟、乱序的原因。




**「数据源延迟乱序」：**属于数据源本身的属性，和下游消费的任务无关。

**「数据加工延迟乱序」：**这是和具体的任务绑定。

其对应关系如下。

| 维度 | 数据源视角（与具体任务无关） | 数据处理任务视角（与具体任务绑定） |
| --- | --- | --- |
| 延迟 | 源日志上报的延迟 | 数据加工过程导致的延迟 |
| 乱序 | 源日志上报的乱序 | 数据加工过程中 shuf... 导致的乱序 |

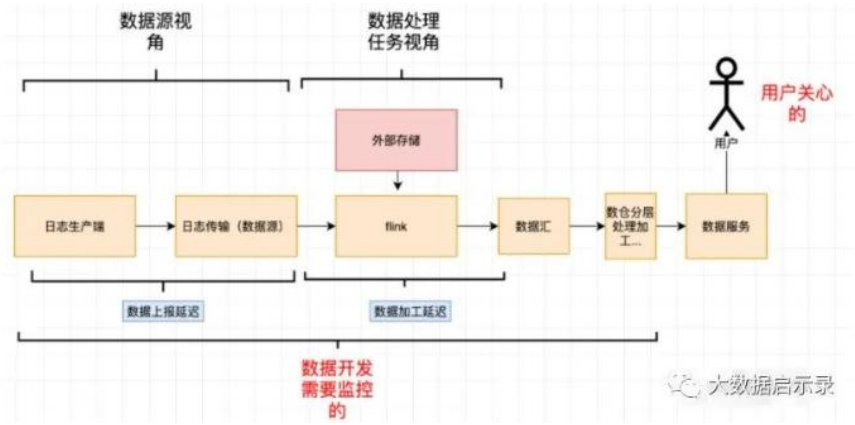
# 数据时延监控

## 整体时延

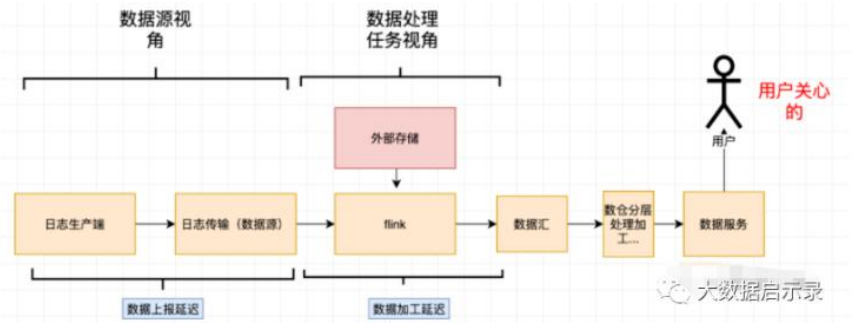
整体时延可以从以下两个角度出发进行计算。

用户视角：只关心最终产出结果时延

开发视角：需要关心整个链路处理时延




## 结果时延监控



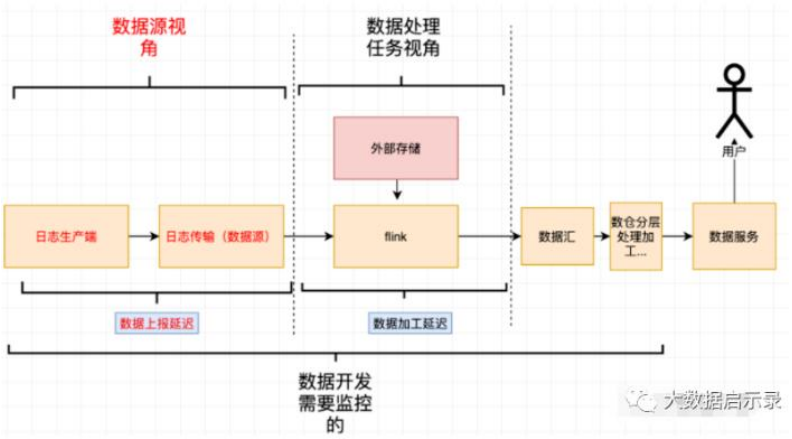

## 监控指标以及报警机制

从用户体验角度直观的反映出数据的整体时延情况

| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | 有数据时效监控中心提供延迟监控 sdk。在看板的 web server 侧将数据时延上报到延迟监控 sdk 中。 |
| | 指标 | 计算 web-server-system-current-timestamp - message-event-timestamp 计算 P99 等指标。 |
| | 优点 | 能从用户体感角度出发，准确的刻画时延情况。 |
| | 缺点 | 对 web server 有埋点侵入性。 |
| 报警 | 机制 | 定时（比如 1min/次） check 监控指标的 P99 指标。 |
| | 阈值 | 判断监控指标的 P99 指标是否超过某个阈值（比如 5 min）。 |
| | 接收人 | 报警反馈给任务链路负责人。 |

## 链路时延监控

### 数据源时延




**这个时延和处理任务无关，单纯从指数据本身的属性，数据本身上报就存在的时延。**

举例：从用户发生消费事件一直到日志进入数据源存储引擎中（比如 kafka），这期间存在的时延。

### 监控指标以及报警机制

| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | 单独有一个任务消费并处理数据源。需要保障这个任务任何时刻都不能有 lag，才能刻画出一个准确的数据源时延情况。 |
| | 指标 | 使用 system-current-timestamp - message-event-timestamp P99 等指标。 |
| | 优点 | 在数据源角度能准确的刻画出数据源事件时间时延情况。 |
| | 缺点 | 为了监控数据源乱序情况，需要单独启动一个任务耗费资源。不建议这种方式进行，如果要做，可以进行采样。而且会侵入用户代码，需要用户指定时间戳 |
| 报警 | 机制 | 定时（比如 1min/次） check 监控指标的 P99 指标 |
| | 阈值 | 判断监控指标的 P99 指标是否超过某个阈值（比如 5 min） |
| | 接收人 | 报警反馈给任务链路负责人 |



上面这种方式是站在数据源视角去精准的衡量出数据延迟情况的，但是很多时候我们只需要在下游任务视角去做这件事会更方便。比如：

| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | 在下游任务处处理数据源时记录数据延迟情况。 |
| | 指标 | 使用任务本地 system-current-timestamp - message-event-timestamp P99 等指标 |
| | 优点 | 节约资源 |
| | 缺点 | 一旦下游任务消费有延迟，我们就不能准确的衡量出数据源的延迟情况了。而且会侵入用户代码，需要用户指定时间戳 |
| 报警 | 机制 | 定时（比如 1min/次） check 监控指标的 P99 指标 |
| | 阈值 | 判断监控指标的 P99 指标是否超过某个阈值（比如 180s） |
| | 接收人 | 报警反馈给任务链路负责人 |

Notes：这里衍生出一个问题，客户端日志数据一般会有以下两种时间戳：

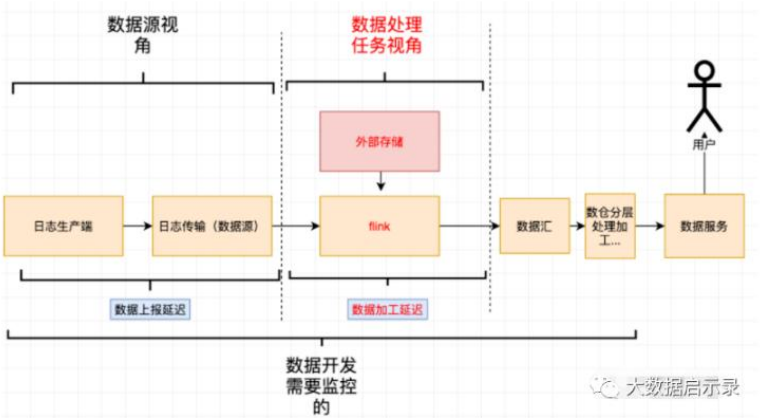
客户端时间戳：用户在客户端操作时的时间戳

服务端时间戳：客户端日志上报到服务端时，日志 server 打上的本地时间戳

因为客户端的软件版本、网络环境、机型、地区的不同，会导致上报的日志「**客户端时间戳**」（用户操作时间戳）的准确性参差不齐（你可能会发现有历史、未来的时间戳）。因此**事件时间都采用服务端时间戳**（日志上报到服务端时，服务端的本地时间戳）来避免这种问题。

当我们采用服务端时间戳时，就基本会发现数据源的时延几乎为 0，因为数据处理链路和日志 server 都是 server 端，因此其之间的数据时延是非常小的，几乎可以忽略不计。

### 数据加工时延




**用于衡量实时任务处理链路的时延。定位链路瓶颈问题。**

### 监控指标以及报警机制

第一个就是 flink 消费数据源的延迟。比如 flink 任务性能不足，产生反压就会有大量 lag。

| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | 在下游任务处处理数据源时记录数据延迟情况。 |
| | 指标 | 用任务本地 system-current-timestamp - kafka-timestamp P99 等指标 |
| | 优点 | 不侵入用户代码 |
| | 缺点 | 可以衡量出任务消费时延情况 |
| 报警 | 机制 | 定时（比如 1min/次） check 监控指标的 P99 指标 |
| | 阈值 | 判断监控指标的 P99 指标是否超过某个阈值（比如 180s） |
| | 接收人 | 报警反馈给任务链路负责人 |



第二部分就是 flink 整个处理过程中的延迟情况:

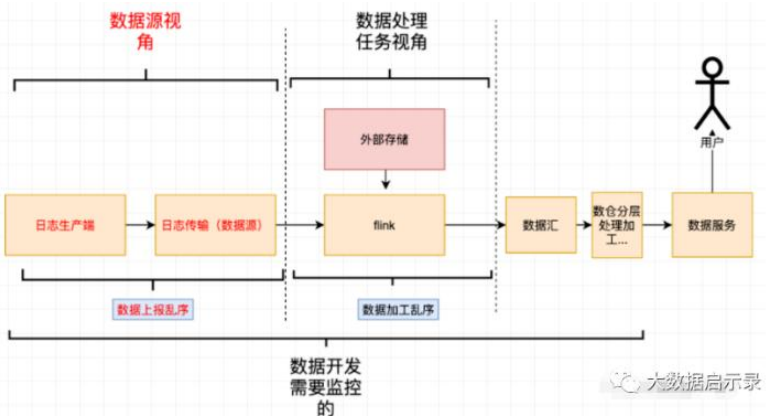
| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | flink 本身自带有 latency marker 机制（详见 flink latency marker） |
| | 指标 | flink latency marker 官方文档 |
| | 优点 | 在下游消费任务的角度准确的刻画出整个 flink 任务加工时延。 |
| | 缺点 | 这个机制会有性能损耗，官方建议只在测试阶段进行使用。这其实已经足够，因为我们在测试阶段就可以基本测试出，flink 任务处理计算的耗时情况 |



## 数据乱序监控

数据乱序监控主要是用来监控数据源、处理任务过程中操作的乱序对产出数据的影响。

### 数据源乱序




指数据本身就存在的乱序，比如客户端网络上报存在的乱序，有的用户在偏远网络较差的地区，所以上报可能就会比很多用户延迟很多，这就造成了数据的乱序。

### 监控指标以及报警机制

| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | 单独有一个任务消费并处理数据源。需要保障这个任务任何时刻都不能有 lag，才能刻画出一个准确的数据源时延情况。 |
| | 指标 | 具体衡量乱序的指标类似于 watermark 分配方式。即为每一个 source consumer 维护一个 max(timestamp)，记为 max_ts，后续来的数据的时间戳记为 cur_tx，如果 cur_tx > max_ts，则说明没有乱序，设置 max_tx = cur_ts，如果出现 cur_ts < max_ts，则说明这条数据发生了乱序，计算出 abs(cur_ts - max_ts) 为具体乱序时长，最终计算乱序时长的 P99 等值。 |
| | 优点 | 在数据源角度能准确的刻画出数据源事件时间乱序情况 |
| | 缺点 | 为了监控数据源乱序情况，需要单独启动一个任务耗费资源。不建议这种方式进行，如果要做，可以进行采样。 |
| 报警 | 机制 | 定时（比如 1min/次） check 监控指标的 P99 指标 |
| | 阈值 | 判断监控指标的 P99 指标是否超过某个阈值（常用 180s） |
| | 接收人 | 报警反馈给任务负责人 |

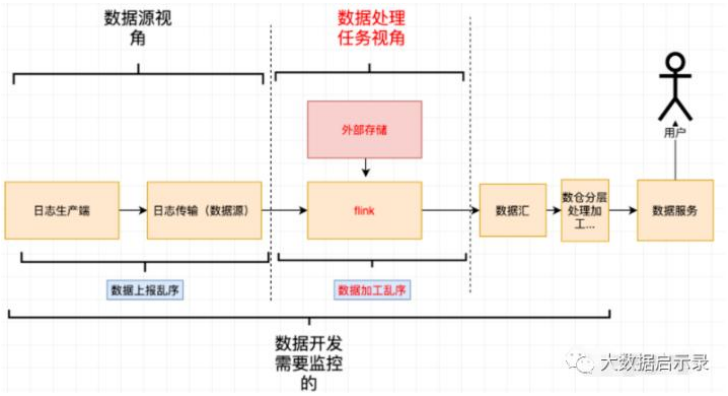
上面这种方式是站在**数据源视**角去精准的衡量出数据乱序情况的，但是很多时候我们只需要在**下游任务视角**去做这件事会更方便。比如：

**监控方式**：在下游任务处处理数据源时记录数据乱序情况。

**监控指标**：衡量指标同上

**虽然数据源可能有乱序，但是这个乱序经过 flink 的一些策略处理后，乱序对计算数据的影响就会被消除。比如用户设置 watermark 时调大 max-out-of-orderness 以及设置 allow-lateness 的处理之后就会解决。**

### 数据加工乱序




单个任务消费上游数据后，内部做一些 rebalance shuffle 操作导致或者加剧数据乱序的情况。从而会导致一些开窗类的任务出现丢数的情况，导致最后数据计算出现误差。

```
DataStream<Model> eventTimeResult = SourceFactory
            .getSourceDataStream(xxx)
            .uid("source")
            .rebalance() // 这里 rebalance 之后会加剧数据乱序，从而可能会导致后续事件时间窗口丢数
            .flatMap(xxx)
            .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor<Model>(Time.minutes(1L)) {
                @Override
                public long extractTimestamp(Model model) {
                    return model.getServerTimestamp();
                }
            })
            .keyBy(KeySelectorFactory.getRemainderKeySelector(xxx))
            .timeWindow(Time.seconds(xxx))
            .process(xxx)
            .uid("process-event-time");
```

### 监控指标以及报警机制

| 板块 | 类型 | 内容 |
| --- | --- | --- |
| 监控 | 方式 | 关心的是乱序最终导致的丢数情况，所以监控丢数条目数即可 |
| | 指标 | Task/Operator numLateRecordsDropped[2] 可以得到由于乱序导致窗口的丢数情况 |
| | 优点 | flink 自带此指标 |
| | 缺点 | |
| 报警 | 机制 | 定时（比如 1min/次） check 监控指标的条目数 |
| | 阈值 | 判断监控指标的条目数是否超过某个阈值（比如 5w 条） |
| | 接收人 | 报警反馈给任务负责人 |



## 上述机制帮助用户暴露出过什么问题

### 1 数据源探查阶段

在数据源探查阶段，通过快速启动数据源消费任务去探查数据源的延迟、乱序程度，确定数据源的可用性。比如发现数据源延迟常年在 5min 以上，那么我们向用户所能保障的数据时延也不会小于 5min。

### 2 暴露延迟、乱序问题

通过我们的实践测试之后，我们发现报警和问题原因是符合 2-8 定律的，甚至比例达到了 2 - 9。即 90% 的问题都可以由 20% 的报警发现。

#### 2.1 90% 的时延问题是由于 flink 任务性能不足导致

报警项：flink 消费 kafka lag 延迟超过 180s

其他监控项辅助定位：flink 任务 cpu 使用率超过 100%；flink 任务 ygc 每分钟超过 20s

#### 2.2 10% 的时延问题是由于数据源延迟导致

报警项：flink 消费 kafka lag 延迟超过 180s；数据源时延超过 180s

其他监控项辅助定位：flink 任务 cpu 使用率正常，每分钟 ygc 时长正常

#### 2.3 90% 的乱序问题是由于数据源乱序导致

报警项：flink 任务窗口算子丢数超过 xx 条；数据源乱序 P99 超过 180s（指 99% 的数据乱序情况不超过 180s）

#### 2.4 10% 的乱序问题是由于 flink 任务加工乱序导致

报警项：flink 任务窗口算子丢数超过 xx 条

他监控项辅助定位：数据源乱序 P99 处于合理范围；并且代码中有 rebalance 操作之后分配 watermark

### 3 确定延迟、乱序问题恢复情况

当我们修复数据延迟、乱序问题之后，我们也需要观察任务的回复情况。上述监控也可以帮助观察问题的恢复情况。比如：延迟、乱序时长变小就说明用户的修复是有效的

# 现状及展望

## 6.1 现状

其实目前很多公司有 flink 消费 kafka lag 时延，Task/Operator numLateRecordsDropped 就已经足够用了。全方位建设上述整个时延监控的成本还是很高的。

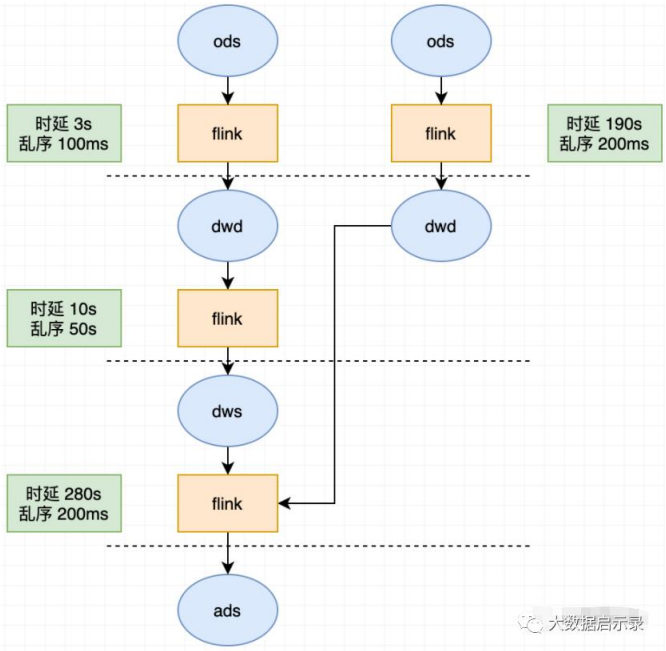
## 6.2 展望

### 6.2.1 实时数据、任务血缘 + 时效性全景图

需求：数仓的上下游链路是很长的，如果想更快快速定位整个数据链路中的时效性问题，就需要一个可视化整体链路时延全局图。

基础能力：需要实时数据、任务血缘（目前想要做到这一点，都已经比较难了，很多大厂的机制都不完善，甚至说没有）

举例：从最终产出的一个 ads 层指标出发，逆推血缘，并展示出时效情况。

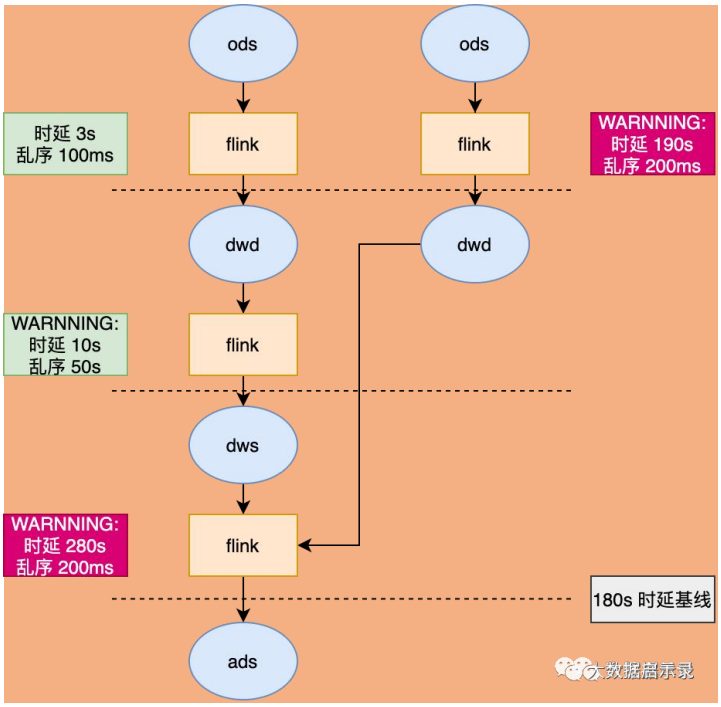


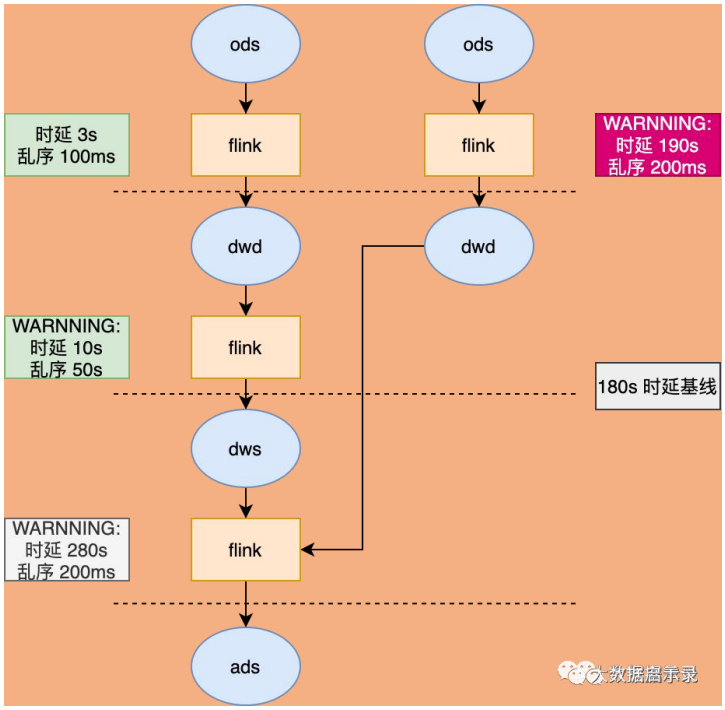

### 6.2.2 实时时效性基线

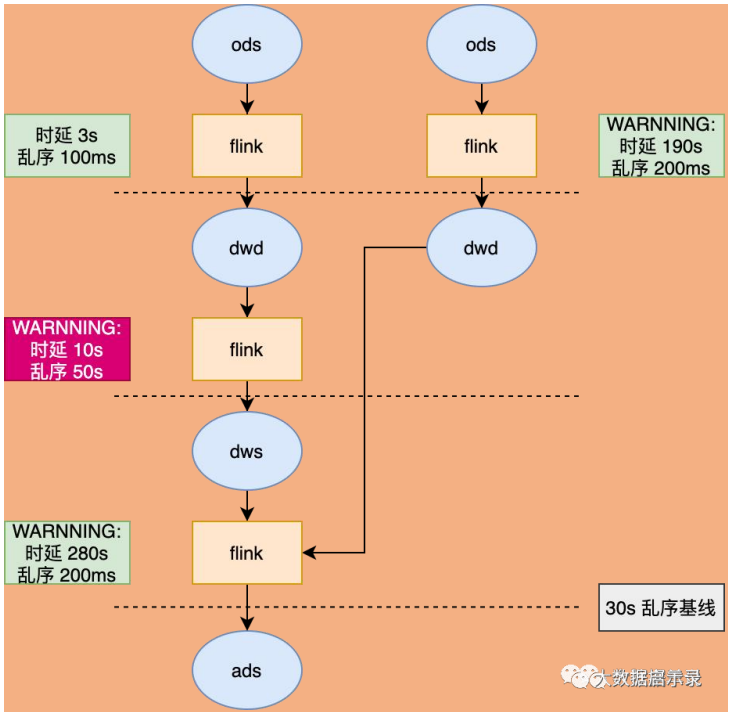
并且将时延超过阈值的链路使用醒目的颜色标注

需求：不同的指标有不同的产出时延标准，有了 6.2.1 的基础能力之后，我们就可以根据具体时延要求设置时效性基线。比如设置最终指标产出时延不能超过 180s。那么基线就是 180s。只要整个链路的产出时延超过 180s 就报警。也可以对某一层的加工链路设置基线。

举例：从最终产出的一个 ads 层指标出发，设置基线 180s，那么下图的任务就可以根据基线设定的任务，逆推计算出链路中时延过长的任务，直接报警。


ods
ods
时延 3s
乱序 100ms
flink
flink
WARNNING:
时延 190s
乱序 200ms
dwd
dwd
WARNNING:
时延 10s
乱序 50s
flink
dws
WARNNING:
时延 280s
乱序 200ms
flink
180s 时延基线
ads
大数据磨汞录


ods
ods
时延 3s
乱序 100ms
flink
flink
WARNNING:
时延 190s
乱序 200ms
dwd
dwd
WARNNING:
时延 10s
乱序 50s
flink
180s 时延基线
dws
WARNNING:
时延 280s
乱序 200ms
flink
ads
大数据磨汞录




补充

### LatencyMarks

与通过水印来标记事件时间的推进进度相似，flink 也用一种特殊的流元素作为延迟的标记，成为 LatencyMarker.LatencyMarker 在 source 上以配置发送间隔，并由任务 task 转发。sink 最后接收到 LatencyMarks，将比较 LatencyMarker 的时间戳与当前系统时间，以确定延迟。LatencyMarker 不会增加作业的延迟，但是 LatencyMarker 与常规记录类似，可以被 delay 阻塞（例如反压情况），因此 LatencyMarker 的延迟与 Record 延迟近似。

**LatencyMarker 是不会向 WaterMarker 一样参与到数据流的用户逻辑中去的，而是直接被各算子转发并统计。**

### metrics.latency.interval

flink 中通过开启配置 **metrics.latency.interval** 来开启 latency 后就可以在 metric 中看到 askManagerJobMetricGroup/operator_id/operator_subtask_index/latency 指标了

metrics.latency.interval 的时间间隔宜大不宜小，在我们的实践中一般配置成 30000（30 秒）左右。一是因为延迟监控的频率可以不用太频繁，二是因为 LatencyMarker 的处理也要消耗时间，只有在 LatencyMarker 的耗时远小于正常 StreamRecord 的耗时时，metric 反映出的数据才贴近实际情况，所以 LatencyMarker 的密度不能太大。

### 粒度

在创建 LatencyStats 之前，先要根据 metrics.latency.granularity 配置项来确定延迟监控的粒度，分为以下 3 档：

single：每个算子单独统计延迟；（跟踪延迟，无需区分:源+源子任务）

operator（默认值）：每个下游算子都统计自己与 Source 算子之间的延迟（跟踪延迟，区分源，但不区分源的子任务）；

subtask：每个下游算子的 sub-task 都统计自己与 Source 算子的 sub-task 之间的延迟。（跟踪延迟，区分源+源子任务）

一般情况下采用默认的 operator 粒度即可，这样在 Sink 端观察到的 latency metric 就是我们最想要的全链路（端到端）延迟，以下也是以该粒度讲解。subtask 粒度太细，会增大所有并行度的负担，不建议使用。

```
SINGLE {
    String createUniqueHistogramName(LatencyMarker marker, OperatorID operatorId, int operatorSubtaskIndex) {
     // 只有自己的 operatorId 和 operatorSubtaskIndex 参与 Metric 名称生成
     // LatencyMarker 带有的 id（源）不参与 Metric 名称生成
     return String.valueOf(operatorId) + operatorSubtaskIndex;
    }
}

OPERATOR {
    String createUniqueHistogramName(LatencyMarker marker, OperatorID operatorId, int operatorSubtaskIndex) {
    // LatencyMarker 带有的 id（源）中 id 参与计算
     return String.valueOf(marker.getOperatorId()) + operatorId + operatorSubtaskIndex;
    }
  }

  SUBTASK {
    String createUniqueHistogramName(LatencyMarker marker, OperatorID operatorId, int operatorSubtaskIndex) {
     return String.valueOf(marker.getOperatorId()) + marker.getSubtaskIndex() + operatorId + operatorSubtaskIndex;
    }
  }
```

任意一个中间 Operator 或 Sink，可以通过配置 metrics.latency.granularity 项，调整与 Source 间统计的粒度（Singe、Operator、Subtask）：

A、统计的时候，可以选择 source 源 id、source 源 subtask index 进行组合，调整统计粒度。

B、统计的时候，当前 Operator 及当前 Operator subtask index 总是参与粒度名称的生成，固定的。

**总结:**

1、LatencyMarker 不参与 window、MiniBatch 的缓存计时，直接被中间 Operator 下发。

2、Metric 路径:TaskManagerJobMetricGroup/operator_id/operator_subtask_index/latency（根据时延配置粒度 Granularity，路径会有变化，参考本文第六章节）

3、每个中间 Operator、以及 Sink 都会统计自己与 Source 节点的链路延迟，我们在监控页面，一般展示 Source 至 Sink 链路延迟。

4、延迟粒度细分到 Task，可以用来排查哪台机器的 Task 时延偏高，进行对比和运维排查。

5、从实现原理来看，发送时延标记间隔配置大一些（例如 20 秒一次），一般不会影响系统处理业务数据的性能（所有的 StreamSource Task 都按间隔发送时延标记，中间节点有多个输出通道的，随机选择一个通道下发，不会复制多份数据出来）。

6、参数配置：metrics.latency.granularityoperator、metrics.latency.history-size、metrics.latency.interval。