

# 尚硅谷大数据之电商实时数仓 之 Flink 总结及优化

(作者：尚硅谷研究院)

版本：V3.0

# 第 1 章 基本原理、常见面试题、重点难点

## 1.1 基本架构

分布式计算引擎、流式、有状态、事件驱动的

Client: 进程

JobManager: 进程

TaskManager: 进程

## 1.2 谈谈你对 slot 的理解?

1) 资源的基本单位: 隔离内存, 不隔离 CPU

2) 可以共享的:

slot 共享组:

不指定, 默认都是"default"组下面的, 所有可以共享

某个算子计算较复杂, 不想共享: 算子.slotSharingGroup("组名") = > 不同组

不会共享同一个 slot

spark 的 excutor 中的 core: task 也是运行在上面, 它是隔离的 cpu 资源, 不隔离内存

## 1.3 什么地方可以设置并行度

配置文件 < 提交参数 < env 全局指定 < 算子单独指定

## 1.4 Flink 的图 Graph

	在哪里生成	传给谁
StreamGraph	Client	Client
JobGraph	Client	JobManager
ExecutionGraph	JobManager	

物理执行图

### ➤ **StreamGraph**

是根据用户通过 Stream API 编写的代码生成的最初的图，用来表示程序的拓扑结构。

### ➤ **JobGraph**

StreamGraph 经过优化后生成了 JobGraph，是提交给 JobManager 的数据结构。主要的优化为：将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。

### ➤ **ExecutionGraph**

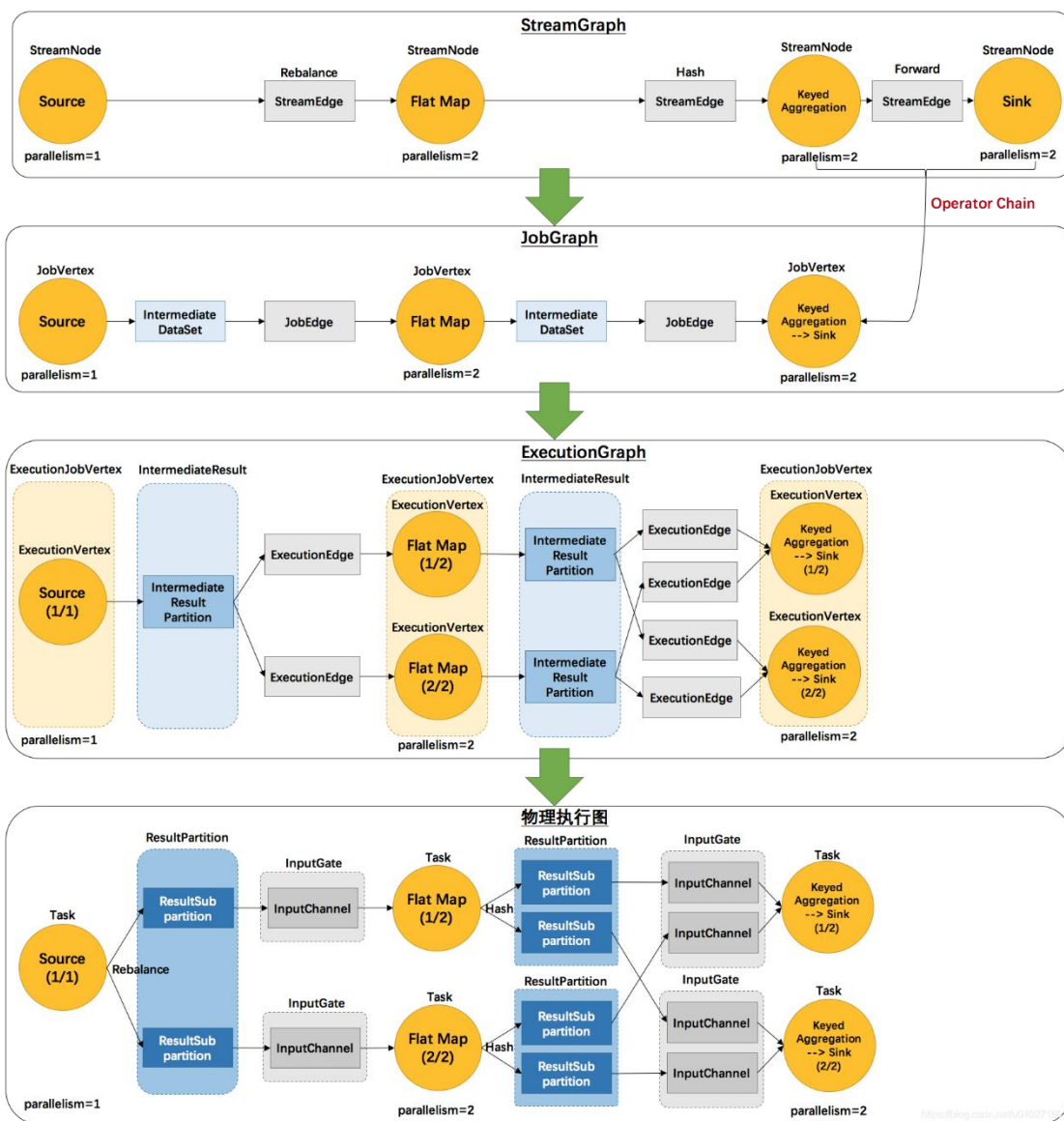
JobManager 根据 JobGraph 生成 ExecutionGraph，ExecutionGraph 是 JobGraph 的并行化版本，是调度层最核心的数据结构。

### ➤ **Physical Graph**

JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个 TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。

2 个并发度（Source 为 1 个并发度）的 SocketTextStreamWordCount 四层执行图的演变过程

```
env.socketTextStream().flatMap(...).keyBy(0).sum(1).print();
```



## 1.5 Flink 与 Spark Streaming 的区别

	Flink	SparkStreaming
计算模型	流式	微批次
窗口	更多、更灵活	没那么多，没那么灵活
状态	有状态的	updatestatebykey
时间语义	三种	处理时间
watermark	有	没有

流式 SQL                  支持                  没有

Spark 和 Flink 的主要差别就在于**计算模型不同**。Spark 采用了微批处理模型，而 Flink 采用了基于操作符的连续流模型。因此，对 Apache Spark 和 Apache Flink 的选择实际上变成了计算模型的选择，而这种选择需要在延迟、吞吐量和可靠性等多个方面进行权衡。

如果企业中非要技术选型从 Spark 和 Flink 这两个主流框架中选择一个来进行流数据处理，我们**推荐使用 Flink**，主（显而）要（易见）的原因为：

- Flink 灵活的窗口
- Exactly Once 语义保证

这两个原因可以大大的解放程序员，加快编程效率，把本来需要程序员花大力气手动完成的工作交给框架完成

## 1.6 Watermark 相关

### 1.6.1 谈谈你对 watermark 的理解

来源：google 的论文，《Streaming 102》、《超越批处理的世界-the dataflow model》

- 1) 逻辑时钟-用来表示事件时间的进展
- 2) 用来解决乱序的问题
- 3) 单调不减的
- 4) 特殊的时间戳，插入到流里面，向下游传播
- 5) 触发窗口、定时器等计算
- 6) Flink 认为，小于 wm 的应该都处理过了，如果还有，称为迟到数据

传递

一对多：广播

多对一：取最小

多对多：复杂的问题简单化，拆分来看。先拆为 1 对多进行广播，然后下游接收是多对一取最小。

## 1.6.2 设置方式

\*forMonotonousTimestamps

\*forBoundedOutOfOrderness

单调递增 forMonotonousTimestamps 是乱序 forBoundedOutOfOrderness 的特殊情况，是子类，延迟为 0

```
output.emitWatermark(new Watermark(maxTimestamp - outOfOrdernessMillis - 1));
```

\*自定义

> 实现 WatermarkGenerator 接口

> 周期性的(推荐) 在 onPeriodicEmit 方法中发射 watermark

默认间隔 autoWatermarkInterval=200 毫秒

通过 env.getConfig().setAutoWatermarkInterval(毫秒数)设置

一般就默认即可，保持在毫秒级，如果设置过大，响应时间就拉长了

> 间歇性的 在 onEvent 方法中发射数据

来一条数据，生成一条 watermark，处理性能下降

## 1.7 Flink 怎么处理乱序、迟到的数据

1) 是用 watermark，设定一个乱序程度

2) 窗口允许迟到

3) 侧输出流

## 1.8 flink 的 join 有几种？ intervaljoin 的底层实现是什么？ join 不上的怎么办？

- 基于窗口

window join

- 基于状态

interval join

- interval join 的底层实现：

底层还是调用的 connect+keyby

核心逻辑：

两条流都初始化了一个 Map 类型的状态，key=时间戳，value=List（数据）

判断当前来的数据是否迟到，如果迟到，直接 return，不处理

不管哪条流的数据来了，都会去对方的 Map 状态里遍历，看能不能匹配上

如果匹配上，就会一起发送给 join 方法

通过注册定时器的方式，超过作用范围，就会清理状态里对应的数据

- join 不上的数据也想要，怎么办？

intervaljoin 只能取到 join 上的数据，类似 sql 里的 inner join

如果需要 join 不上的数据，就不要用 interval join：

api: coGroup+connect

sql: left join 、 right join

## 1.9 keyby 的实现原理

两次 hash

第一次 hash =》 key.hashCode() => hash1

第二次 hash => murmurhash(hash1) => hash2

计算数据如何发放下游的分区：

hash2 \* 下游算子的并行度 / 最大并行度(默认 128)

## 1.10 分区、分组、分流

分区：物理资源，也就是算子的一个并行实例

分组：逻辑上的划分

分流：侧输出流来分流

# 第 2 章 生产相关问题

## 2.1 你们 Flink 是基于什么模式部署的

Standalone ： flink 自己管资源

Yarn ： yarn 管资源

session

per-job ： 主要用这个

application

K8S

Mesos



## 2.2 yarn 的 per-job 和 session 能不能同时存在

能，对于 yarn，只是一个 application

## 2.3 flink 的集群规模多大

基于 yarn 部署的，确定了并行度、每个 TaskManager 的 slot 数

=》动态申请 TaskManager

理论上，只要是 yarn 的节点，都可以被使用，具体多少，指定 2 个参数后，由 yarn 决定

## 2.4 你们资源怎么分配的

### ➤ 内存

JobManager：一般 2G 够了

TaskManager：

确定单个 TM 的 slot 数 =》单个 TM 的 slot 数：核数 = 1：1 或 2：1（没有

很多计算逻辑，etl）

比如，1 个 TM 上有 4 核，建议分配 4 个 slot

=》每个 slot 能有 1~2G 内存

=》单个 TM 给 4~8G

特定的任务 ==》存储了很大的状态，比如 20G、单个 TM 可能要大于 20G

### ➤ 并行度

粗略设置：全局设置为 kafka 的分区数

精细设置：

source: kafka, 等于分区数

transform: 如果计算复杂, 建议给 2 的 n 次方 ==》 8、16

sink: kafka, 等于分区数

- 估算一个 job 需要大概占用的内存

老大 2G

小弟: 每个 TM 给 4 个 slot, 平均并行度是 8, 需要 2 小弟, 每个小弟给 4G ==》 8G

所以, 平均需要 10G

## 2.5 窗口相关

- 分类:

时间: 滚动、滑动、会话

条数: 滚动、滑动

- 四个组成:

窗口分配器

触发器

驱逐器

计算函数

- 窗口是怎么划分的, 以事件时间窗口为例

$start = ts - (ts - offset + window\_size) \% window\_size$  ==》 时间戳向下取整

$end = start + window\_size$

- 窗口为什么是左闭右开?

$maxTs = end - 1ms$

➤ 窗口是何时触发输出

时间进展 大于等于 窗口的最大时间戳

时间进展  $\geq \text{maxTs}$

➤ 窗口的生命周期

创建：属于本窗口的第一条数据来的时候，new 的窗口对象，放到一个单例集合里

销毁：时间进展 达到了 窗口的最大时间戳 + 允许迟到的时间

时间进展  $\geq \text{maxTs} + \text{allowedLateness}$

➤ 窗口时间和 Watermark 关系

\*如果事件时间是顺序的，那么  $\text{watermark} = \text{eventtime} - 1\text{ms}$

\*如果事件时间是乱序的，那么  $\text{watermark} = \text{eventtime} - \text{最大乱序时间} - 1\text{ms}$

\*如果设置了允许迟到时间  $\text{allowedLateness}$

> 每次一个元素进来，会读取这个元素的事件时间并计算出当前这个元素对应的 watermark，然后根据 wm 判断是否触发操作以及关闭窗口；如果长时间没有，不会一直无限等，可以设置允许迟到时间，那么

> 当  $\text{wm} \geq \text{窗口结束时间}$  的时候，不会关闭窗口，只会触发计算

> 当  $\text{wm} \geq \text{窗口结束时间} + \text{允许迟到时间}$  的时候，关闭窗口

> 当  $\text{窗口结束时间} \leq \text{wm} < \text{窗口结束时间} + \text{允许迟到时间}$  的时候，每来一条迟到数据，就会计算一次。

> 如果到了允许迟到时间，还有数据过来，可以设置侧输出流  
`ds.setOutputLateData(new OutputTag("侧输出流名"))`

➤ 读取文件时窗口的计算规则

如果读取文件，属于有界流，到文件的末尾后，文件最后的数据可能没有触发窗口的计算，Flink 会自动在数据流中添加一个 Long 的最大值作为 WaterMark，保证文件的数据全部处理

➤ 为什么所有窗口同时被 Long 的最大值触发？

因为默认周期 200ms，读取完数据，还没有更新 watermark，默认是 Long 的最小值；

数据读取完后，watermark 马上被更新为 Long 的最大值，一下子把所有窗口触发了

## 2.6 Flink 是怎么保证端到端的一致性（重点）

source 端：可重发

flink：依赖于 checkpoint 机制 -》Chandy-Lamport 算法

sink 端：幂等性、事务性 (2pc)

## 2.7 检查点相关

### 2.7.1 介绍 checkpoint 原理、流程

➤ 检查点就是状态的一个副本、备份，只不过这里的状态包含 source、operation 以及 sink 阶段

➤ 通过 `env.enableCheckpointing(1000, CheckpointingMode.EXACTLY_ONCE)` 开启检查点，并设置周期

➤ 通过异步分界线快照 (asynchronous barrier snapshotting)。该算法大致基于 Chandy-Lamport 分布式快照算法

➤ 思路

- jobmanager 触发 checkpoint 操作, barrier 从 source 开始向下传递, source 中遇到 barrier 的算子将状态(读取数据的 offset)存入状态后端, 并通知 jobmanager
- 中间计算算子同样保存状态(计算结果)到 checkpoint
- sink 操作也一样将状态保存到 checkpoint, 这里仅仅是一个标记, 标记整个端到端操作完毕。这里涉及二阶段提交

➤ 注意:

- barrier 向下游传递, 计算任务会等待所有的输入分区的 barrier 都到达才会计算, 这种处理称为分界线对齐
- 可以设置超时时间
- 对于 barrier 已经到达的分区, 如果还有数据过来, 会对当前数据进行缓存, 因为这个数据不属于这个检查点
- 对于 barrier 还没有到达的分区, 如果有数据过来, 数据会被正常处理, 接收到 barrier 之后, 将状态保存到检查点。
- Checkpoint Barrier 对齐机制实现 Exactly-Once 语义。如果 Barrier 不对齐, 即 At Least Once 语义。
- barrier 对齐: 等待上游所有实例, 将同一编号的 barrier 都传递到了, 才会触发自己的备份

精准一次: 先到的 barrier, 后面的数据, 缓存在缓冲区, 不处理

至少一次: 先到的 barrier, 后面的数据, 不缓存, 接着处理

➤ 企业怎么选:

如果是准确性要求非常高的, 比如与钱相关的, 设置为精准一次

如果是对准确性要求不高，对性能有要求，可以设置为 至少一次

## 2.7.2 savepoint 与 checkpoint 区别，使用场景

他们算法都一样

savepoint 是手动的、checkpoint 是自动的

升级代码、升级框架

命令：

触发 savepoint：

`flink savepoint 指定 jobid hdfs 路径`

`flink cancel -s 指定 jobid hdfs 路径` =》手动取消作业的同时，进行保存点的执行

从 savepoint 恢复：

`flink run -s HDFS 路径`

## 2.7.3 你们 flink 的 checkpoint 都设置了什么

ck 间隔： 建议一般 3~10 分钟，(注意，写入 kafka 用了 2pc，导致延迟)

语义级别： 精准一次、至少一次

超时时间： 一般参考间隔，3 分钟

失败次数： 一般 5 次

最小等待间隔： 参考间隔，3 分钟

保留 checkpoint：

重启策略：

固定延迟重启： 重启次数、重启的间隔

故障率重启：在时间范围内的重启次数、重启间隔、重启的时间范围

企业怎么选？

默认固定延迟重启，建议故障率更好，因为固定延迟会累加

## 2.8 维表 join 方案

- 1) 预加载维表：比如，在 open 里，定期查询 mysql
- 2) 热存储：将维表数据放在外部数据库：mysql、hbase
  - a) 异步 io
  - b) 缓存
- 3) 广播维表：能保证拿到的是新的维度数据，但是维度数据都在中状态里

## 第 3 章 资源配置调优

Flink 性能调优的第一步，就是为任务分配合适的资源，在一定范围内，增加资源的分配与性能的提升是成正比的，实现了最优的资源配置后，在此基础上再考虑进行后面论述的性能调优策略。

提交方式主要是 **yarn-per-job**，资源的分配在使用脚本提交 Flink 任务时进行指定。

标准的 Flink 任务提交脚本（Generic CLI 模式），从 1.11 开始，增加了通用客户端模式，参数使用 -D <property=value> 指定

```
bin/flink run \  
-t yarn-per-job \  
-d \  
-p 5 \ 指定并行度  
-Dyarn.application.queue=test \ 指定 yarn 队列  
-Djobmanager.memory.process.size=1024mb \ 指定 JM 的总进程大小  
-Dtaskmanager.memory.process.size=1024mb \ 指定每个 TM 的总进程大小  
-Dtaskmanager.numberOfTaskSlots=2 \ 指定每个 TM 的 slot 数  
-c com.atguigu.app.dwd.LogBaseApp \  

```

```
/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar
```

参数列表：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/deployment/config.html>

## 3.1 内存设置 (1CPU 配置 4G 内存)

生产资源配置：

```
bin/flink run \  
-t yarn-per-job \  
-d \  
-p 5 \ 指定并行度  
-Dyarn.application.queue=test \ 指定 yarn 队列  
-Djobmanager.memory.process.size=2048mb \ JM2~4G 足够  
-Dtaskmanager.memory.process.size=6144mb \ 单个 TM2~8G 足够  
-Dtaskmanager.numberOfTaskSlots=2 \ 与容器核数 1core: 1slot 或 1core: 2slot  
-c com.atguigu.app.dwd.LogBaseApp \  
/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Flink 是实时流处理，关键在于资源情况能不能抗住高峰时期每秒的数据量，通常用 QPS/TPS 来描述数据情况。

## 3.2 并行度设置

### 3.2.1 最优并行度计算

开发完成后，先进行压测。任务并行度给 10 以下，测试单个并行度的处理上限。然后

总 QPS/单并行度的处理能力 = 并行度

不能只从 QPS 去得出并行度，因为有些字段少、逻辑简单的任务，单并行度一秒处理几万条数据。而有些数据字段多，处理逻辑复杂，单并行度一秒只能处理 1000 条数据。

最好根据高峰期的 QPS 压测，并行度\*1.2 倍，富余一些资源。



### 3.2.2 Source 端并行度的配置

数据源端是 Kafka，Source 的并行度设置为 Kafka 对应 Topic 的分区数。

如果已经等于 Kafka 的分区数，消费速度仍跟不上数据生产速度，考虑下 Kafka 要扩大分区，同时调大并行度等于分区数。

Flink 的一个并行度可以处理一至多个分区的数据，如果并行度多于 Kafka 的分区数，那么就会造成有的并行度空闲，浪费资源。

### 3.2.3 Transform 端并行度的配置

#### ➤ Keyby 之前的算子

一般不会做太重的操作，都是比如 map、filter、flatMap 等处理较快的算子，并行度可以和 source 保持一致。

#### ➤ Keyby 之后的算子 (KeyGroup 最小值为 128)

如果并发较大，建议设置并行度为 2 的整数次幂，例如：128、256、512；

小并发任务的并行度不一定需要设置成 2 的整数次幂；

大并发任务如果没有 KeyBy，并行度也无需设置为 2 的整数次幂；

### 3.2.4 Sink 端并行度的配置

Sink 端是数据流向下游的地方，可以根据 Sink 端的数据量及下游的服务抗压能力进行评估。

如果 Sink 端是 Kafka，可以设为 Kafka 对应 Topic 的分区数。

Sink 端的数据量小，比较常见的就是监控告警的场景，并行度可以设置的小一些。

Source 端的数据量是最小的，拿到 Source 端流过来的数据后做了细粒度的拆分，数

据量不断的增加, 到 Sink 端的数据量就非常大。那么在 Sink 到下游的存储中间件的时候就需要提高并行度。

另外 Sink 端要与下游的服务进行交互, 并行度还得根据下游的服务抗压能力来设置, 如果在 Flink Sink 这端的数据量过大的话, 且 Sink 处并行度也设置的很大, 但下游的服务完全撑不住这么大的并发写入, 可能会造成下游服务直接被写挂, 所以最终还是要在 Sink 处的并行度做一定的权衡。

### 3.3 RocksDB 大状态调优

RocksDB 是基于 LSM Tree 实现的 (类似 HBase), 写数据都是先缓存到内存中, 所以 RocksDB 的写请求效率比较高。RocksDB 使用内存结合磁盘的方式来存储数据, 每次获取数据时, 先从内存中 blockcache 中查找, 如果内存中没有再去磁盘中查询。优化后差不多单并行度 TPS 5000 record/s, 性能瓶颈主要在于 RocksDB 对磁盘的读请求, 所以当处理性能不够时, 仅需要横向扩展并行度即可提高整个 Job 的吞吐量。以下几个调优参数:

#### ➤ 设置本地 RocksDB 多目录

在 flink-conf.yaml 中配置:

```
state.backend.rocksdb.localdir:  
/data1/flink/rocksdb,/data2/flink/rocksdb,/data3/flink/rocksdb
```

**注意:** 不要配置单块磁盘的多个目录, 务必将目录配置到多块不同的磁盘上, 让多块磁盘来分担压力。当设置多个 RocksDB 本地磁盘目录时, Flink 会随机选择要使用的目录, 所以就可能存在三个并行度共用同一目录的情况。如果服务器磁盘数较多, 一般不会出现该情况, 但是如果任务重启后吞吐量较低, 可以检查是否发生了多个并行度共用同一块磁盘的情况。

当一个 TaskManager 包含 3 个 slot 时, 那么单个服务器上的三个并行度都对磁盘造成频繁读写, 从而导致三个并行度的之间相互争抢同一个磁盘 io, 这样务必导致三个并

行度的吞吐量都会下降。设置多目录实现三个并行度使用不同的硬盘从而减少资源竞争。

如下所示是测试过程中磁盘的 IO 使用率，可以看出三个大状态算子的并行度分别对应了三块磁盘，这三块磁盘的 IO 平均使用率都保持在 45% 左右，IO 最高使用率几乎都是 100%，而其他磁盘的 IO 平均使用率相对低很多。由此可见使用 RocksDB 做为状态后端且有大状态的频繁读取时，对磁盘 IO 性能消耗确实比较大。

	last	min	avg	max
disk.io.util/device=sda	1.891	0.000	6.817	44.334
disk.io.util/device=sdb	5.274	1.491	14.723	98.708
disk.io.util/device=sdg	10.348	0.397	11.285	46.215
disk.io.util/device=sdd	60.100	8.557	47.072	100.000
disk.io.util/device=sdi	48.159	14.641	44.091	100.000
disk.io.util/device=sdj	97.811	18.272	48.237	99.602

如下图所示，其中两个并行度共用了 sdb 磁盘，一个并行度使用 sdj 磁盘。可以看到 sdb 磁盘的 IO 使用率已经达到了 91.6%，就会导致 sdb 磁盘对应的两个并行度吞吐量大大降低，从而使得整个 Flink 任务吞吐量降低。如果每个服务器上有一两块 SSD，强烈建议将 RocksDB 的本地磁盘目录配置到 SSD 的目录下，从 HDD 改为 SSD 对于性能的提升可能比配置 10 个优化参数更有效。

	last	min	avg	max
disk.io.util/device=sda	0.000	0.000	5.524	44.080
disk.io.util/device=sdb	98.205	65.672	91.656	100.000
disk.io.util/device=sdg	3.490	0.000	22.013	95.821
disk.io.util/device=sdd	29.611	0.100	12.050	78.963
disk.io.util/device=sdi	2.792	0.000	11.915	79.860
disk.io.util/device=sdj	42.871	5.279	35.530	100.000
disk.io.util/device=sdj	8.475	0.299	14.172	100.000

- **state.backend.incremental**: 开启增量检查点，默认 false，改为 true。
- **state.backend.rocksdb.predefined-options**:  
  
SPINNING\_DISK\_OPTIMIZED\_HIGH\_MEM 设置为机械硬盘+内存模式，有条件上 SSD，指定为 FLASH\_SSD\_OPTIMIZED
- **state.backend.rocksdb.block.cache-size**: 整个 RocksDB 共享一个 block

cache, 读数据时内存的 cache 大小, 该参数越大读数据时缓存命中率越高, 默认大小为 8 MB, 建议设置到 64 ~ 256 MB。

- **state.backend.rocksdb.thread.num:** 用于后台 flush 和合并 sst 文件的线程数, 默认为 1, 建议调大, 机械硬盘用户可以改为 4 等更大的值。
- **state.backend.rocksdb.writebuffer.size:** RocksDB 中, 每个 State 使用一个 Column Family, 每个 Column Family 使用独占的 write buffer, 建议调大, 例如: 32M
- **state.backend.rocksdb.writebuffer.count:** 每个 Column Family 对应的 writebuffer 数目, 默认值是 2, 对于机械磁盘来说, 如果内存足够大, 可以调大到 5 左右
- **state.backend.rocksdb.writebuffer.number-to-merge:** 将数据从 writebuffer 中 flush 到磁盘时, 需要合并的 writebuffer 数量, 默认值为 1, 可以调成 3。
- **state.backend.local-recovery:** 设置本地恢复, 当 Flink 任务失败时, 可以基于本地的状态信息进行恢复任务, 可能不需要从 hdfs 拉取数据

## 3.4 Checkpoint 设置

一般我们的 Checkpoint 时间间隔可以设置为分钟级别, 例如 1 分钟、3 分钟, 对于状态很大的任务每次 Checkpoint 访问 HDFS 比较耗时, 可以设置为 5~10 分钟一次 Checkpoint, 并且调大两次 Checkpoint 之间的暂停间隔, 例如设置两次 Checkpoint 之间至少暂停 4 或 8 分钟。

如果 Checkpoint 语义配置为 EXACTLY\_ONCE, 那么在 Checkpoint 过程中还会存在 barrier 对齐的过程, 可以通过 Flink Web UI 的 Checkpoint 选项卡来查看

Checkpoint 过程中各阶段的耗时情况，从而确定到底是哪个阶段导致 Checkpoint 时间过长然后针对性的解决问题。

RocksDB 相关参数在 1.3 中已说明，可以在 flink-conf.yaml 指定，也可以在 Job 的代码中调用 API 单独指定，这里不再列出。

```
// 使用 RocksDBStateBackend 做为状态后端，并开启增量 Checkpoint
RocksDBStateBackend rocksDBStateBackend = new
RocksDBStateBackend("hdfs://hadoop102:8020/flink/checkpoints", true);
env.setStateBackend(rocksDBStateBackend);

// 开启 Checkpoint，间隔为 3 分钟
env.enableCheckpointing(TimeUnit.MINUTES.toMillis(3));
// 配置 Checkpoint
CheckpointConfig checkpointConf = env.getCheckpointConfig();
checkpointConf.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
// 最小间隔 4 分钟
checkpointConf.setMinPauseBetweenCheckpoints(TimeUnit.MINUTES.toMillis(4))
// 超时时间 10 分钟
checkpointConf.setCheckpointTimeout(TimeUnit.MINUTES.toMillis(10));
// 保存 checkpoint
checkpointConf.enableExternalizedCheckpoints(
CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

## 3.5 使用 Flink ParameterTool 读取配置

在实际开发中，有各种环境（开发、测试、预发、生产），作业也有很多的配置：算子的并行度配置、Kafka 数据源的配置（broker 地址、topic 名、group.id）、Checkpoint 是否开启、状态后端存储路径、数据库地址、用户名和密码等各种各样的配置，可能每个环境的这些配置对应的值都是不一样的。

如果你是直接在代码里面写死的配置，每次换个环境去运行测试作业，都要重新去修改代码中的配置，然后编译打包，提交运行，这样就要花费很多时间在这些重复的劳动力上了。在 Flink 中可以通过使用 ParameterTool 类读取配置，它可以读取环境变量、运行参数、配置文件。

ParameterTool 是可序列化的，所以你可以将它当作参数进行传递给算子的自定义函数。

数类。

## 3.5.1 读取运行参数

我们可以在 Flink 的提交脚本添加运行参数，格式：

➤ --参数名 参数值

➤ -参数名 参数值

在 Flink 程序中可以直接使用 `ParameterTool.fromArgs(args)` 获取到所有的参数，

也可以通过 `parameterTool.get("username")` 方法获取某个参数对应的值。

举例：通过运行参数指定 jobname

```
bin/flink run \  
-t yarn-per-job \  
-d \  
-p 5 \ 指定并行度  
-Dyarn.application.queue=test \ 指定 yarn 队列  
-Djobmanager.memory.process.size=1024mb \ 指定 JM 的总进程大小  
-Dtaskmanager.memory.process.size=1024mb \ 指定每个 TM 的总进程大小  
-Dtaskmanager.numberOfTaskSlots=2 \ 指定每个 TM 的 slot 数  
-c com.atguigu.app.dwd.LogBaseApp \  
/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar \  
--jobname dwd-LogBaseApp //参数名自己随便起，代码里对应上即可
```

在代码里获取参数值：

```
ParameterTool parameterTool = ParameterTool.fromArgs(args);  
String myJobname = parameterTool.get("jobname"); //参数名对应  
env.execute(myJobname);
```

## 3.5.2 读取系统属性

`ParameterTool` 还支持通过 `ParameterTool.fromSystemProperties()` 方法读取系

统属性。做个打印：

```
ParameterTool parameterTool = ParameterTool.fromSystemProperties();  
System.out.println(parameterTool.toMap().toString());
```

可以得到全面的系统属性，部分结果：

```
{sun.desktop=windows, awt.toolkit=sun.awt.windows.WToolkit, file.encoding.pkg=sun.io, java.specification.version=1.8, sun.cpu.isalist=amd64, sun.jnu.encoding=GBK, java.class.path=D  
, java.vm.specification.vendor=Oracle Corporation, java.specification.name=Java Platform API Specification, java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment, sun.boot.class.pa
```

### 3.5.3 读取配置文件

可以使用 `ParameterTool.fromPropertiesFile("/application.properties")` 读取 `properties` 配置文件。可以将所有要配置的地方（比如并行度和一些 Kafka、MySQL 等配置）都写成可配置的，然后其对应的 `key` 和 `value` 值都写在配置文件中，最后通过 `ParameterTool` 去读取配置文件获取对应的值。

### 3.5.4 注册全局参数

在 `ExecutionConfig` 中可以将 `ParameterTool` 注册为全作业参数的参数，这样就可以被 `JobManager` 的 web 端以及用户自定义函数中以配置值的形式访问。

```
StreamExecutionEnvironment env =  
StreamExecutionEnvironment.getExecutionEnvironment();  
env.getConfig().setGlobalJobParameters(ParameterTool.fromArgs(args));
```

可以不用将 `ParameterTool` 当作参数传递给算子的自定义函数，直接在用户自定义的 `Rich` 函数中直接获取到参数值了。

```
env.addSource(new RichSourceFunction() {  
    @Override  
    public void run(SourceContext sourceContext) throws Exception {  
        while (true) {  
            ParameterTool parameterTool =  
(ParameterTool) getRuntimeContext().getExecutionConfig().getGlobalJobParameters();  
        }  
    }  
}  
  
@Override  
public void cancel() {  
}  
})
```

## 3.6 压测方式

压测的方式很简单，先在 kafka 中积压数据，之后开启 Flink 任务，出现反压，就是处理瓶颈。相当于水库先积水，一下子泄洪。

数据可以是自己造的模拟数据，也可以是生产中的部分数据。

## 第 4 章 反压处理

反压 (BackPressure) 通常产生于这样的场景：短时间的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。许多日常问题都会导致反压，例如，垃圾回收停顿可能会导致流入的数据快速堆积，或遇到大促、秒杀活动导致流量陡增。反压如果不能得到正确的处理，可能会导致资源耗尽甚至系统崩溃。

反压机制是指系统能够自己检测到被阻塞的 Operator，然后自适应地降低源头或上游数据的发送速率，从而维持整个系统的稳定。Flink 任务一般运行在多个节点上，数据从上游算子发送到下游算子需要网络传输，若系统在反压时想要降低数据源头或上游算子数据的发送速率，那么肯定也需要网络传输。所以下面先来了解一下 Flink 的网络流控 (Flink 对网络数据流量的控制) 机制。

### 4.1 反压现象及定位

Flink 的反压太过于天然了，导致无法简单地通过监控 BufferPool 的使用情况来判断反压状态。Flink 通过对运行中的任务进行采样来确定其反压，如果一个 Task 因为反压导致处理速度降低了，那么它肯定会卡在向 LocalBufferPool 申请内存块上。那么该 Task 的 stack trace 应该是这样：

```
java.lang.Object.wait(Native Method)
o.a.f.[...].LocalBufferPool.requestBuffer(LocalBufferPool.java:163)
o.a.f.[...].LocalBufferPool.requestBufferBlocking(LocalBufferPool.java:133) [...]
```

监控对正常的任务运行有一定影响，因此只有当 Web 页面切换到 Job 的 BackPressure 页面时，JobManager 才会对该 Job 触发反压监控。默认情况下，JobManager 会触发 100 次 stack trace 采样，每次间隔 50ms 来确定反压。Web 界面看到的比率表示在内部方法调用中有多少 stack trace 被卡在



LocalBufferPool.requestBufferBlocking(), 例如: 0.01 表示在 100 个采样中只有 1 个被卡在 LocalBufferPool.requestBufferBlocking()。采样得到的比例与反压状态的对应关系如下:

- OK:  $0 \leq \text{比例} \leq 0.10$
- LOW:  $0.10 < \text{比例} \leq 0.5$
- HIGH:  $0.5 < \text{比例} \leq 1$

Task 的状态为 OK 表示没有反压, HIGH 表示这个 Task 被反压。

### 4.1.1 利用 Flink Web UI 定位产生反压的位置

在 Flink Web UI 中有 BackPressure 的页面, 通过该页面可以查看任务中 subtask 的反压状态, 如下两图所示, 分别展示了状态是 OK 和 HIGH 的场景。

排查的时候, 先把 operator chain 禁用, 方便定位。

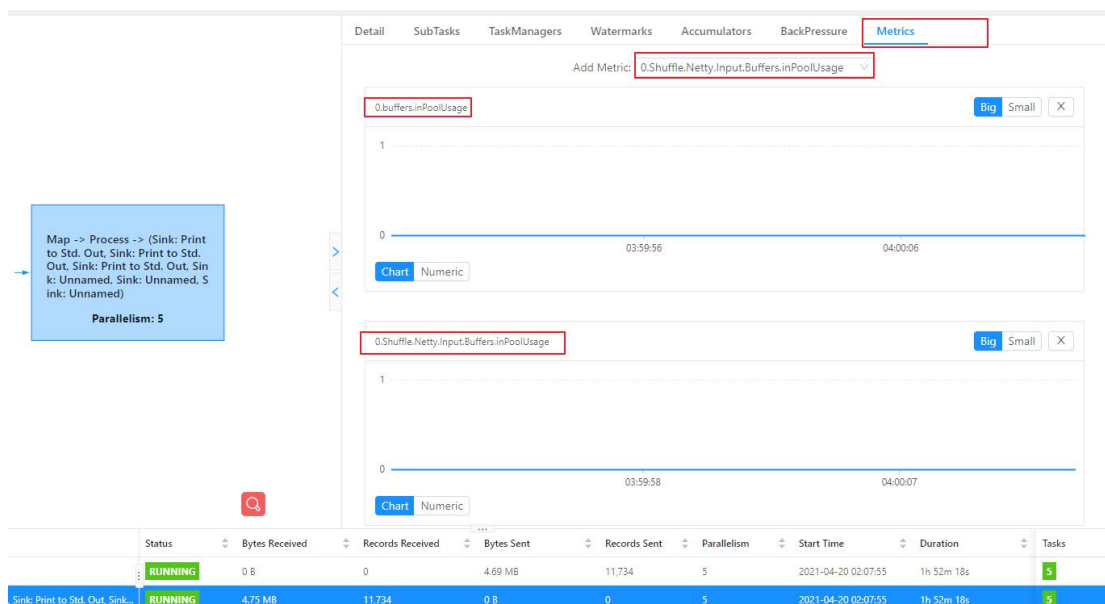
Detail	SubTasks	TaskManagers	Watermarks	Accumulators	BackPressure	Metrics
				Measurement: 17s ago    Back Pressure Status: <span>OK</span>		
SubTask			Ratio	Status		
1			0.01	<span>OK</span>		
2			0	<span>OK</span>		
3			0	<span>OK</span>		
>			0	<span>OK</span>		
5			0	<span>OK</span>		
6			0.01	<span>OK</span>		
7			0	<span>OK</span>		
8			0	<span>OK</span>		

Detail	SubTasks	TaskManagers	Watermarks	Accumulators	BackPressure	Metrics
				Measurement: 1m 8s ago	Back Pressure Status:	HIGH
SubTask			Ratio			Status
1			1			HIGH
2			1			HIGH
3			1			HIGH
>			1			HIGH
5			0.97			HIGH
6			1			HIGH
7			1			HIGH
8			1			HIGH

## 4.1.2 利用 Metrics 定位反压位置

当某个 Task 吞吐量下降时,基于 Credit 的反压机制,上游不会给该 Task 发送数据,所以该 Task 不会频繁卡在向 Buffer Pool 去申请 Buffer。反压监控实现原理就是监控 Task 是否卡在申请 buffer 这一步,所以遇到瓶颈的 Task 对应的反压页面必然会显示 OK,即表示没有受到反压。

如果该 Task 吞吐量下降,造成该 Task 上游的 Task 出现反压时,必然会存在:该 Task 对应的 InputChannel 变满,已经申请不到可用的 Buffer 空间。如果该 Task 的 InputChannel 还能申请到可用 Buffer,那么上游就可以给该 Task 发送数据,上游 Task 也就不会被反压了,所以说遇到瓶颈且导致上游 Task 受到反压的 Task 对应的 InputChannel 必然是满的(这里不考虑网络遇到瓶颈的情况)。从这个思路出发,可以对该 Task 的 InputChannel 的使用情况进行监控,如果 InputChannel 使用率 100%,那么该 Task 就是我们要找的反压源。Flink 1.9 及以上版本 inPoolUsage 表示 inputFloatingBuffersUsage 和 inputExclusiveBuffersUsage 的总和。



反压时，可以看到遇到瓶颈的该 Task 的 inPoolUsage 为 1。

## 4.2 反压的原因及处理

先检查基本原因，然后再深入研究更复杂的原因，最后找出导致瓶颈的原因。下面列出从最基本到比较复杂的一些反压潜在原因。

注意：反压可能是暂时的，可能是由于负载高峰、CheckPoint 或作业重启引起的数据积压而导致反压。如果反压是暂时的，应该忽略它。另外，请记住，断断续续的反压会影响我们分析和解决问题。

### 4.2.1 系统资源

检查涉及服务器基本资源的使用情况，如 CPU、网络或磁盘 I/O，目前 Flink 任务使用最主要的还是内存和 CPU 资源，本地磁盘、依赖的外部存储资源以及网卡资源一般都不会是瓶颈。如果某些资源被充分利用或大量使用，可以借助分析工具，分析性能瓶颈（JVM Profiler+ FlameGraph 生成火焰图）。

如何生成火焰图：<http://www.54tianzhisheng.cn/2020/10/05/flink-jvm-profiler/>

如何读懂火焰图: <https://zhuanlan.zhihu.com/p/29952444>

- 针对特定的资源调优 Flink
- 通过增加并行度或增加集群中的服务器数量来横向扩展
- 减少瓶颈算子上游的并行度, 从而减少瓶颈算子接收的数据量 (不建议, 可能造成整个 Job 数据延迟增大)

## 4.2.2 垃圾收集 (GC)

长时间 GC 暂停会导致性能问题。可以通过打印调试 GC 日志 (通过 `-XX:+PrintGCDetails`) 或使用某些内存或 GC 分析器 (GCViewer 工具) 来验证是否处于这种情况。

- 在 Flink 提交脚本中, 设置 JVM 参数, 打印 GC 日志:

```
bin/flink run \  
-t yarn-per-job \  
-d \  
-p 5 \ 指定并行度  
-Dyarn.application.queue=test \ 指定 yarn 队列  
-Djobmanager.memory.process.size=1024mb \ 指定 JM 的总进程大小  
-Dtaskmanager.memory.process.size=1024mb \ 指定每个 TM 的总进程大小  
-Dtaskmanager.numberOfTaskSlots=2 \ 指定每个 TM 的 slot 数  
-Denv.java.opts="-XX:+PrintGCDetails -XX:+PrintGCDateStamps"  
-c com.atguigu.app.dwd.LogBaseApp \  
/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.  
jar
```

- 下载 GC 日志的方式:

因为是 on yarn 模式, 运行的节点一个一个找比较麻烦。可以打开 WebUI, 选择 JobManager 或者 TaskManager, 点击 Stdout, 即可看到 GC 日志, 点击下载按钮即可将 GC 日志通过 HTTP 的方式下载下来。

Metrics	Configuration	Logs	Stdout	Log List
1	2021-04-20T04:52:29.547+0800:	[GC (Allocation Failure) [PSYoungGen: 27136K->4080K(31232K)] 27136K->4080K(102400K), 0.0070905 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]		
2	2021-04-20T04:52:29.836+0800:	[GC (Allocation Failure) [PSYoungGen: 31224K->4076K(31232K)] 31224K->4076K(102400K), 0.0042281 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]		
3	2021-04-20T04:52:30.075+0800:	[GC (Allocation Failure) [PSYoungGen: 31218K->4076K(31232K)] 31218K->4076K(102400K), 0.0048259 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]		
4	2021-04-20T04:52:30.267+0800:	[GC (Metadata GC Threshold) [PSYoungGen: 23775K->4076K(31232K)] 29039K->12050K(102400K), 0.0052357 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]		
5	2021-04-20T04:52:30.272+0800:	[Full GC (Metadata GC Threshold) [PSYoungGen: 4076K->0K(31232K)] [ParOldGen: 7974K->6756K(71168K)] 12050K->6756K(102400K), [Metaspace: 20743K->20743K(1067000K)], 0.0273359 secs] [Times: user=0.05 sys=0.01, real=0.03 secs]		
6	2021-04-20T04:52:30.334+0800:	[GC (Allocation Failure) [PSYoungGen: 27136K->4077K(31232K)] 33092K->10009K(102400K), 0.0024189 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]		
7	2021-04-20T04:52:30.716+0800:	[GC (Allocation Failure) [PSYoungGen: 31213K->4004K(23040K)] 38045K->11930K(94200K), 0.0035638 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]		
8	2021-04-20T04:52:30.809+0800:	[GC (Allocation Failure) [PSYoungGen: 23038K->3640K(27136K)] 38074K->11476K(98304K), 0.0051239 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]		
9	2021-04-20T04:52:30.972+0800:	[GC (Allocation Failure) [PSYoungGen: 22584K->4814K(27648K)] 38420K->12657K(98816K), 0.0071002 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]		
10	2021-04-20T04:52:31.078+0800:	[GC (Allocation Failure) [PSYoungGen: 24270K->3125K(27136K)] 32113K->13085K(98304K), 0.0060212 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]		
11	2021-04-20T04:52:31.224+0800:	[GC (Allocation Failure) [PSYoungGen: 22581K->4069K(27648K)] 32541K->15309K(98816K), 0.0028261 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]		
12	2021-04-20T04:52:31.341+0800:	[GC (Metadata GC Threshold) [PSYoungGen: 22670K->1693K(27136K)] 33018K->13557K(98304K), 0.0021736 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]		
13	2021-04-20T04:52:31.343+0800:	[Full GC (Metadata GC Threshold) [PSYoungGen: 1693K->0K(27136K)] [ParOldGen: 11864K->9804K(71168K)] 13557K->9804K(98304K), [Metaspace: 34750K->34741K(1079200K)], 0.0360412 secs] [Times: user=0.09 sys=0.00, real=0.04 secs]		
14	2021-04-20T04:52:31.546+0800:	[GC (Allocation Failure) [PSYoungGen: 19456K->3273K(27136K)] 29260K->13005K(98304K), 0.0027952 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]		
15	2021-04-20T04:52:31.679+0800:	[GC (Allocation Failure) [PSYoungGen: 22217K->2370K(27136K)] 32029K->13455K(98304K), 0.0023721 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]		
16	2021-04-20T04:52:31.754+0800:	[GC (Allocation Failure) [PSYoungGen: 21314K->3180K(27648K)] 32299K->14265K(98816K), 0.0026041 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]		
17	2021-04-20T04:52:31.835+0800:	[GC (Allocation Failure) [PSYoungGen: 22636K->2979K(27136K)] 33720K->14095K(98304K), 0.0025775 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]		
18	2021-04-20T04:52:31.966+0800:	[GC (Allocation Failure) [PSYoungGen: 22435K->2812K(27648K)] 34451K->15402K(98816K), 0.0067992 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]		
19	2021-04-20T04:52:38.081+0800:	[PSYoungGen: 22780K->1796K(27648K)] 35370K->15848K(98816K), 0.0037455 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]		
20	2021-04-20T04:52:38.152+0800:	[GC (Allocation Failure) [PSYoungGen: 21750K->2057K(27648K)] 35009K->16116K(98816K), 0.0023396 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]		
21	2021-04-20T04:52:38.795+0800:	[GC (Allocation Failure) [PSYoungGen: 22025K->3040K(27648K)] 36084K->17766K(98816K), 0.0170811 secs] [Times: user=0.02 sys=0.01, real=0.02 secs]		
22	2021-04-20T04:52:43.317+0800:	[GC (Allocation Failure) [PSYoungGen: 23017K->5303K(27136K)] 37734K->20240K(98304K), 0.0060959 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]		
23	2021-04-20T04:52:46.446+0800:	[GC (Allocation Failure) [PSYoungGen: 25345K->5834K(26112K)] 40211K->20700K(97200K), 0.0070362 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]		
24	2021-04-20T04:54:09.808+0800:	[GC (Allocation Failure) [PSYoungGen: 25802K->4966K(27136K)] 40670K->19848K(98304K), 0.0177395 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]		
25				

## ➤ 分析 GC 日志：

通过 GC 日志分析出单个 Flink Taskmanager 堆总大小、年轻代、老年代分配的内存空间、Full GC 后老年代剩余大小等，相关指标定义可以去 Github 具体查看。

GCViewer 地址：<https://github.com/chewiebug/GCViewer>

扩展：最重要的指标是 **Full GC 后，老年代剩余大小**这个指标，按照《Java 性能优化权威指南》这本书 Java 堆大小计算法则，设 Full GC 后老年代剩余大小空间为  $M$ ，那么堆的大小建议  $3 \sim 4$  倍  $M$ ，新生代为  $1 \sim 1.5$  倍  $M$ ，老年代应为  $2 \sim 3$  倍  $M$ 。

## 4.2.3 CPU/线程瓶颈

有时，一个或几个线程导致 CPU 瓶颈，而整个机器的 CPU 使用率仍然相对较低，则可能无法看到 CPU 瓶颈。例如，48 核的服务器上，单个 CPU 瓶颈的线程仅占用 2% 的 CPU 使用率，就算单个线程发生了 CPU 瓶颈，我们也看不出来。可以考虑使用 2.2.1 提到的分析工具，它们可以显示每个线程的 CPU 使用情况来识别热线程。

## 4.2.4 线程竞争

与上面的 CPU/线程瓶颈问题类似，subtask 可能会因为共享资源上高负载线程的竞争而成为瓶颈。同样，可以考虑使用 2.2.1 提到的分析工具，考虑在用户代码中查找同步开

销、锁竞争，尽管避免在用户代码中添加同步。

## 4.2.5 负载不平衡

如果瓶颈是由数据倾斜引起的，可以尝试通过将数据分区的 key 进行加盐或通过实现本地预聚合来减轻数据倾斜的影响。（关于数据倾斜的详细解决方案，会在下一章节详细讨论）

## 4.2.6 外部依赖

如果发现我们的 Source 端数据读取性能比较低或者 Sink 端写入性能较差，需要检查第三方组件是否遇到瓶颈。例如，Kafka 集群是否需要扩容，Kafka 连接器是否并行度较低，HBase 的 rowkey 是否遇到热点问题。关于第三方组件的性能问题，需要结合具体的组件来分析。

# 第 5 章 数据倾斜

## 5.1 判断是否存在数据倾斜

相同 Task 的多个 Subtask 中，个别 Subtask 接收到的数据量明显大于其他 Subtask 接收到的数据量，通过 Flink Web UI 可以精确地看到每个 Subtask 处理了多少数据，即可判断出 Flink 任务是否存在数据倾斜。通常，数据倾斜也会引起反压。

<	Detail	SubTasks	TaskManagers	W
ID	Bytes Received	Records Received		
4	1.74 GB	45,570,996		
8	101 MB	2,566,528		
9	57.5 MB	1,171,285		
7	49.0 MB	930,769		
> 3	27.5 MB	583,279		
< 0	8.75 MB	218,105		
1	7.76 MB	168,709		
5	228 KB	2		
2	228 KB	2		

## 5.2 数据倾斜的解决

### 5.2.1 keyBy 之前发生数据倾斜

如果 keyBy 之前就存在数据倾斜，上游算子的某些实例可能处理的数据较多，某些实例可能处理的数据较少，产生该情况可能是因为数据源的**数据本身就不均匀**，例如由于某些原因 Kafka 的 topic 中某些 partition 的数据量较大，某些 partition 的数据量较少。对于不存在 keyBy 的 Flink 任务也会出现该情况。

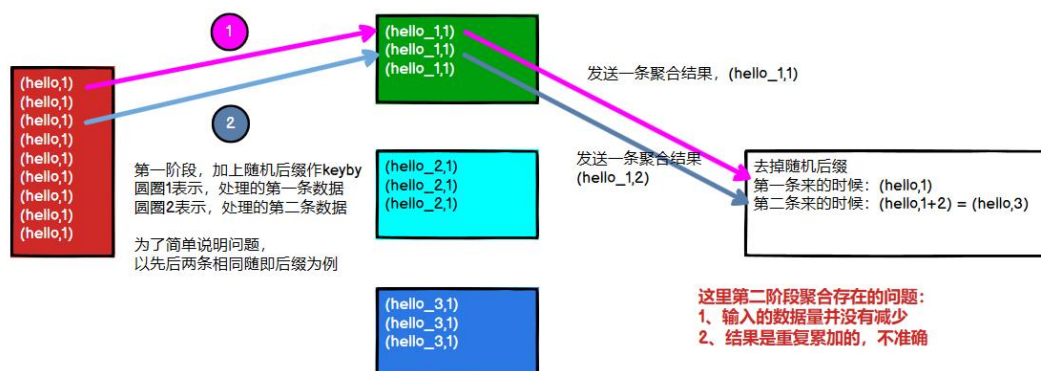
这种情况，需要让 Flink 任务强制进行 shuffle。使用 shuffle、**rebalance** 或 rescale 算子即可将数据均匀分配，从而解决数据倾斜的问题。

### 5.2.2 keyBy 后的聚合操作存在数据倾斜

使用 **LocalKeyBy** 的**思想**：在 keyBy 上游算子数据发送之前，首先在上游算子的本地对数据进行聚合后再发送到下游，使下游接收到的数据量大大减少，从而使得 keyBy 之后的聚合操作不再是任务的瓶颈。类似 MapReduce 中 Combiner 的思想，**但是这要求聚合操作必须是多条数据或者一批数据才能聚合，单条数据没有办法通过聚合来减少数据量。**从 Flink LocalKeyBy 实现原理来讲，必然会存在一个**积攒批次**的过程，在上游算子中必须

攒够一定的数据量，对这些数据聚合后再发送到下游。

**注意：**Flink 是实时流处理，如果 keyby 之后的聚合操作存在数据倾斜，且没有开窗口的情况下，简单的认为使用两阶段聚合，是不能解决问题的。因为这个时候 Flink 是来一条处理一条，且向下游发送一条结果，对于原来 keyby 的维度（第二阶段聚合）来讲，数据量并没有减少，且结果重复计算（非 FlinkSQL，未使用回撤流），如下图所示：



➤ **实现方式：**以计算 PV 为例，keyby 之前，使用 flatMap 实现 LocalKeyby

```
class LocalKeyByFlatMap extends RichFlatMapFunction<String, Tuple2<String,
//Checkpoint 时为了保证 Exactly Once, 将 buffer 中的数据保存到该 ListState 中
private ListState<Tuple2<String, Long>> localPvStatListState;

//本地 buffer, 存放 local 端缓存的 app 的 pv 信息
private HashMap<String, Long> localPvStat;

//缓存的数据量大小, 即: 缓存多少数据再向下游发送
private int batchSize;

//计数器, 获取当前批次接收的数据量
private AtomicInteger currentSize;

//构造器, 批次大小传参
LocalKeyByFlatMap(int batchSize){
    this.batchSize = batchSize;
}

@Override
public void flatMap(String in, Collector collector) throws Exception {
    // 将新来的数据添加到 buffer 中
    Long pv = localPvStat.getOrDefault(in, 0L);
    localPvStat.put(in, pv + 1);
    // 如果到达设定的批次, 则将 buffer 中的数据发送到下游
    if(currentSize.incrementAndGet() >= batchSize){
        // 遍历 Buffer 中数据, 发送到下游
        for(Map.Entry<String, Long> appIdPv: localPvStat.entrySet()) {
```



```

        collector.collect(Tuple2.of(appIdPv.getKey(),
appIdPv.getValue())
    }
    // Buffer 清空, 计数器清零
    localPvStat.clear();
    currentSize.set(0);
}
}

@Override
public void snapshotState(FunctionSnapshotContext functionSnapshotConte
    // 将 buffer 中的数据保存到状态中, 来保证 Exactly Once
    localPvStatListState.clear();
    for(Map.Entry<String, Long> appIdPv: localPvStat.entrySet()) {
        localPvStatListState.add(Tuple2.of(appIdPv.getKey(), appIdPv.ge
    }
}

@Override
public void initializeState(FunctionInitializationContext context) {
    // 从状态中恢复 buffer 中的数据
    localPvStatListState = context.getOperatorStateStore().getListState
    new ListStateDescriptor<>("localPvStat",
    TypeInformation.of(new TypeHint<Tuple2<String, Long>>{}));
    localPvStat = new HashMap();
    if(context.isRestored()) {
        // 从状态中恢复数据到 localPvStat 中
        for(Tuple2<String, Long> appIdPv: localPvStatListState.get()){
            long pv = localPvStat.getOrDefault(appIdPv.f0, 0L);
            // 如果出现 pv != 0, 说明改变了并行度,
            // ListState 中的数据会被均匀分发到新的 subtask 中
            // 所以单个 subtask 恢复的状态中可能包含两个相同的 app 的数据
            localPvStat.put(appIdPv.f0, pv + appIdPv.f1);
        }
        // 从状态恢复时, 默认认为 buffer 中数据量达到了 batchSize, 需要向下游发
        currentSize = new AtomicInteger(batchSize);
    } else {
        currentSize = new AtomicInteger(0);
    }
}
}
}

```

### 5.2.3 keyBy 后的窗口聚合操作存在数据倾斜

因为使用了窗口, 变成了有界数据的处理 (3.2.2 已分析过), 窗口默认是触发时才会输出一条结果发往下游, 所以可以使用**两阶段聚合**的方式:

**实现思路:**

- 第一阶段聚合：key 拼接随机数前缀或后缀，进行 keyby、开窗、聚合

**注意：**聚合完不再是 *WindowedStream*，要获取 *WindowEnd* 作为窗口标记作为第二

阶段分组依据，避免不同窗口的结果聚合到一起)

- 第二阶段聚合：去掉随机数前缀或后缀，按照原来的 key 及 windowEnd 作 keyby、聚合

## 第 6 章 KafkaSource 调优

### 6.1 动态发现分区

当 *FlinkKafkaConsumer* 初始化时，每个 subtask 会订阅一批 partition，但是当 Flink 任务运行过程中，如果被订阅的 topic 创建了新的 partition，*FlinkKafkaConsumer* 如何实现动态发现新创建的 partition 并消费呢？

在使用 *FlinkKafkaConsumer* 时，可以开启 partition 的动态发现。通过 Properties 指定参数开启（单位是毫秒）：

```
FlinkKafkaConsumerBase.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS
```

该参数表示间隔多久检测一次是否有新创建的 partition。默认值是 Long 的最小值，表示不开启，大于 0 表示开启。开启时会启动一个线程根据传入的 interval 定期获取 Kafka 最新的元数据，新 partition 对应的那一个 subtask 会自动发现并从 earliest 位置开始消费，新创建的 partition 对其他 subtask 并不会产生影响。

代码如下所示：

```
properties.setProperty(FlinkKafkaConsumerBase.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS, 30 * 1000 + "");
```

## 6.2 从 Kafka 数据源生成 watermark

Kafka 单分区内有序，多分区间无序。在这种情况下，可以使用 Flink 中可识别 Kafka 分区的 watermark 生成机制。使用此特性，将在 Kafka 消费端内部针对每个 Kafka 分区生成 watermark，并且不同分区 watermark 的合并方式与在数据流 shuffle 时的合并方式相同。

在单分区内有序的情况下，使用时间戳单调递增按分区生成的 watermark 将生成完美的全局 watermark。

可以不使用 TimestampAssigner，直接用 Kafka 记录自身的时间戳：

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

Properties properties = new Properties();
properties.setProperty("bootstrap.servers",
"hadoop1:9092,hadoop2:9092,hadoop3:9092");
properties.setProperty("group.id", "ffffffffffff");

FlinkKafkaConsumer<String> kafkaSourceFunction = new FlinkKafkaConsumer<>(
    "flinktest",
    new SimpleStringSchema(),
    properties
);

kafkaSourceFunction.assignTimestampsAndWatermarks(
    WatermarkStrategy
        .forBoundedOutOfOrderness(Duration.ofMinutes(2))
);

env.addSource(kafkaSourceFunction)
```

## 6.3 设置空闲等待

如果数据源中的某一个分区/分片在一段时间内未发送事件数据，则意味着 WatermarkGenerator 也不会获得任何新数据去生成 watermark。我们称这类数据源为空闲输入或空闲源。在这种情况下，当某些其他分区仍然发送事件数据的时候就会出现问题的。比如 Kafka 的 Topic 中，由于某些原因，造成个别 Partition 一直没有新的数据。

由于下游算子 watermark 的计算方式是取所有不同的上游并行数据源 watermark 的最小值，则其 watermark 将不会发生变化，导致窗口、定时器等不会被触发。

为了解决这个问题，你可以使用 WatermarkStrategy 来检测空闲输入并将其标记为空闲状态。

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

Properties properties = new Properties();
properties.setProperty("bootstrap.servers",
    "hadoop1:9092,hadoop2:9092,hadoop3:9092");
properties.setProperty("group.id", "ffffffffffff");

FlinkKafkaConsumer<String> kafkaSourceFunction = new FlinkKafkaConsumer<>(
    "flinktest",
    new SimpleStringSchema(),
    properties
);

kafkaSourceFunction.assignTimestampsAndWatermarks(
    WatermarkStrategy
        .forBoundedOutOfOrderness(Duration.ofMinutes(2))
        .withIdleness(Duration.ofMinutes(5))
);

env.addSource(kafkaSourceFunction)
```

## 6.4 Kafka 的 offset 消费策略

FlinkKafkaConsumer 可以调用以下 API，注意与“auto.offset.reset”区分开：

- setStartFromGroupOffsets(): 默认消费策略，默认读取上次保存的 offset 信息，如果是应用第一次启动，读取不到上次的 offset 信息，则会根据这个参数 auto.offset.reset 的值来进行消费数据。建议使用这个。
- setStartFromEarliest(): 从最早的数据开始进行消费，忽略存储的 offset 信息
- setStartFromLatest(): 从最新的数据进行消费，忽略存储的 offset 信息
- setStartFromSpecificOffsets(Map): 从指定位置进行消费
- setStartFromTimestamp(long): 从 topic 中指定的时间点开始消费，指定时间点之前的数据忽略

- 当 checkpoint 机制开启的时候, KafkaConsumer 会定期把 kafka 的 offset 信息还有其他 operator 的状态信息一块保存起来。当 job 失败重启的时候, Flink 会从最近一次的 checkpoint 中进行恢复数据, 重新从保存的 offset 消费 kafka 中的数据 (也就是说, 上面几种策略, 只有第一次启动的时候起作用)。
- 为了能够使用支持容错的 kafka Consumer, 需要开启 checkpoint

## 第 7 章 FlinkSQL 调优

FlinkSQL 官网配置参数:

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/config.html>

### 7.1 Group Aggregate 优化

#### 7.1.1 开启 MiniBatch (提升吞吐)

MiniBatch 是微批处理, 原理是缓存一定的数据后再触发处理, 以减少对 State 的访问, 从而提升吞吐并减少数据的输出量。MiniBatch 主要依靠在每个 Task 上注册的 Timer 线程来触发微批, 需要消耗一定的线程调度性能。

- **MiniBatch 默认关闭, 开启方式如下:**

```
// 初始化 table environment
TableEnvironment tEnv = ...

// 获取 tableEnv 的配置对象
Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数:
// 开启 miniBatch
configuration.setString("table.exec.mini-batch.enabled", "true");
// 批量输出的间隔时间
configuration.setString("table.exec.mini-batch.allow-latency", "5s");
// 防止 oom 设置每个批次最多缓存数据的条数, 可以设为 2 万条
configuration.setString("table.exec.mini-batch.size", "20000");
```

- FlinkSQL 参数配置列表：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/config.html>

- 适用场景

微批处理通过增加延迟换取高吞吐，如果有超低延迟的要求，不建议开启微批处理。通常对于聚合的场景，微批处理可以显著的提升系统性能，建议开启。

- **注意事项：**

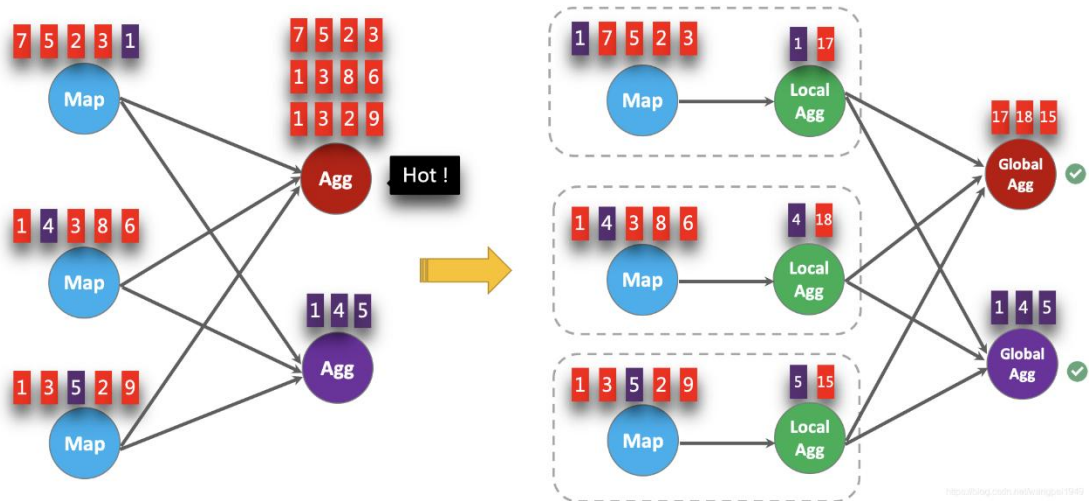
- 1) 目前，key-value 配置项仅被 **Blink planner** 支持。
- 2) 1.12 之前的版本有 bug，开启 miniBatch，不会清理过期状态，也就是说如果设置状态的 TTL，无法清理过期状态。1.12 版本才修复这个问题。

参考 ISSUE: <https://issues.apache.org/jira/browse/FLINK-17096>

## 7.1.2 开启 LocalGlobal（解决常见数据热点问题）

LocalGlobal 优化将原先的 Aggregate 分成 Local+Global 两阶段聚合，即 MapReduce 模型中的 Combine+Reduce 处理模式。第一阶段在上游节点本地攒一批数据进行聚合（localAgg），并输出这次微批的增量值（Accumulator）。第二阶段再将收到的 Accumulator 合并（Merge），得到最终的结果（GlobalAgg）。

LocalGlobal 本质上能够靠 LocalAgg 的聚合筛除部分倾斜数据，从而降低 GlobalAgg 的热点，提升性能。结合下图理解 LocalGlobal 如何解决数据倾斜的问题。



由上图可知：

- 未开启 LocalGlobal 优化, 由于流中的数据倾斜, Key 为红色的聚合算子实例需要处理更多的记录, 这就导致了热点问题。
- 开启 LocalGlobal 优化后, 先进行本地聚合, 再进行全局聚合。可大大减少 GlobalAgg 的热点, 提高性能。

#### ➤ LocalGlobal 开启方式：

1) LocalGlobal 优化需要先开启 MiniBatch, 依赖于 MiniBatch 的参数。

2) `table.optimizer.agg-phase-strategy`: 聚合策略。默认 AUTO, 支持参数 AUTO、

TWO\_PHASE(使用 LocalGlobal 两阶段聚合)、ONE\_PHASE(仅使用 Global 一阶段聚合)。

```
// 初始化 table environment
TableEnvironment tEnv = ...

// 获取 tableEnv 的配置对象
Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数:
// 开启 miniBatch
configuration.setString("table.exec.mini-batch.enabled", "true");
// 批量输出的间隔时间
configuration.setString("table.exec.mini-batch.allow-latency", "5 s");
// 防止 OOM 设置每个批次最多缓存数据的条数, 可以设为 2 万条
configuration.setString("table.exec.mini-batch.size", "20000");
// 开启 LocalGlobal
configuration.setString("table.optimizer.agg-phase-strategy",
    "TWO_PHASE");
```

➤ 判断是否生效

观察最终生成的拓扑图的节点名字中是否包含 GlobalGroupAggregate 或 LocalGroupAggregate。

➤ 适用场景

LocalGlobal 适用于提升如 SUM、COUNT、MAX、MIN 和 AVG 等普通聚合的性能，以及解决这些场景下的数据热点问题。

➤ **注意事项：**

- 1) 需要先开启 MiniBatch
- 2) 开启 LocalGlobal 需要 UDAF 实现 Merge 方法。

### 7.1.3 开启 Split Distinct（解决 COUNT DISTINCT 热点问题）

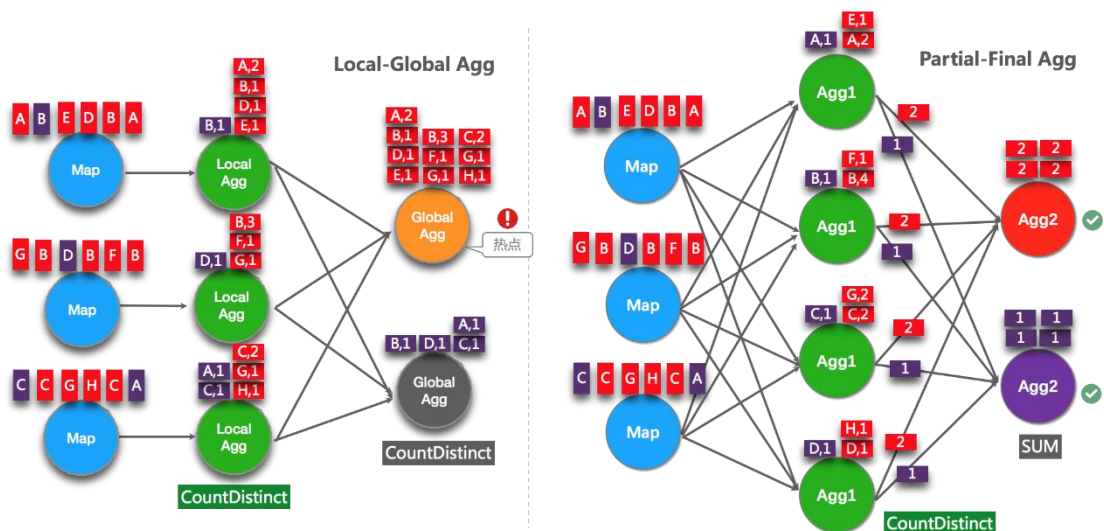
LocalGlobal 优化针对普通聚合（例如 SUM、COUNT、MAX、MIN 和 AVG）有较好的效果，对于 COUNT DISTINCT 收效不明显，因为 COUNT DISTINCT 在 Local 聚合时，对于 DISTINCT KEY 的去重率不高，导致在 Global 节点仍然存在热点。

之前，为了解决 COUNT DISTINCT 的热点问题，通常需要手动改写为两层聚合（增加按 Distinct Key 取模的打散层）。

从 Flink1.9.0 版本开始，提供了 COUNT DISTINCT 自动打散功能，不需要手动重写。

Split Distinct 和 LocalGlobal 的原理对比参见下图。





## 举例：统计一天的 UV

```
SELECT day, COUNT(DISTINCT user_id)
FROM T
GROUP BY day
```

## 如果手动实现两阶段聚合：

```
SELECT day, SUM(cnt)
FROM (
  SELECT day, COUNT(DISTINCT user_id) as cnt
  FROM T
  GROUP BY day, MOD(HASH_CODE(user_id), 1024)
)
GROUP BY day
```

第一层聚合：将 Distinct Key 打散求 COUNT DISTINCT。

第二层聚合：对打散去重后的数据进行 SUM 汇总。

## ➤ Split Distinct 开启方式

默认不开启，使用参数显式开启：

- `table.optimizer.distinct-agg.split.enabled: true`，默认 `false`。
- `table.optimizer.distinct-agg.split.bucket-num`: Split Distinct 优化在第一层聚合中，被打散的 bucket 数目。默认 1024。

```
// 初始化 table environment
TableEnvironment tEnv = ...

// 获取 tableEnv 的配置对象
Configuration configuration = tEnv.getConfig().getConfiguration();
```

```
// 设置参数:
// 开启 Split Distinct
configuration.setString("table.optimizer.distinct-agg.split.enabled",
"true");
// 第一层打散的 bucket 数目
configuration.setString("table.optimizer.distinct-agg.split.bucket-num",
"1024");
```

➤ 判断是否生效

观察最终生成的拓扑图的节点名中是否包含 Expand 节点, 或者原来一层的聚合变成了两层的聚合。

➤ 适用场景

使用 COUNT DISTINCT, 但无法满足聚合节点性能要求。

➤ **注意事项:**

- 1) 目前不能在包含 UDAF 的 Flink SQL 中使用 Split Distinct 优化方法。
- 2) 拆分出来的两个 GROUP 聚合还可参与 LocalGlobal 优化。
- 3) 从 Flink1.9.0 版本开始, 提供了 COUNT DISTINCT **自动打散**功能, 不需要手动重

写 (不用像上面的例子去手动实现)。

## 7.1.4 改写为 AGG WITH FILTER 语法 (提升大量 COUNT DISTINCT 场景性能)

在某些场景下, 可能需要从不同维度来统计 UV, 如 Android 中的 UV, iPhone 中的 UV, Web 中的 UV 和总 UV, 这时, 可能会使用如下 CASE WHEN 语法。

```
SELECT
  day,
  COUNT(DISTINCT user_id) AS total_uv,
  COUNT(DISTINCT CASE WHEN flag IN ('android', 'iphone') THEN user_id ELSE NULL
END) AS app_uv,
  COUNT(DISTINCT CASE WHEN flag IN ('wap', 'other') THEN user_id ELSE NULL END)
AS web_uv
FROM T
GROUP BY day
```

在这种情况下, 建议使用 FILTER 语法, 目前的 Flink SQL 优化器可以识别同一唯一键

上的不同 FILTER 参数。如，在上面的示例中，三个 COUNT DISTINCT 都作用在 user\_id 列上。此时，经过优化器识别后，Flink 可以只使用一个共享状态实例，而不是三个状态实例，可减少状态的大小和对状态的访问。

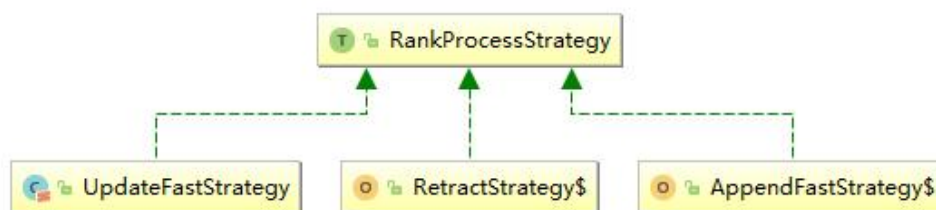
将上边的 CASE WHEN 替换成 FILTER 后，如下所示:

```
SELECT
  day,
  COUNT(DISTINCT user_id) AS total_uv,
  COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('android', 'iphone')) AS
  app_uv,
  COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('wap', 'other')) AS web_uv
FROM T
GROUP BY day
```

## 7.2 TopN 优化

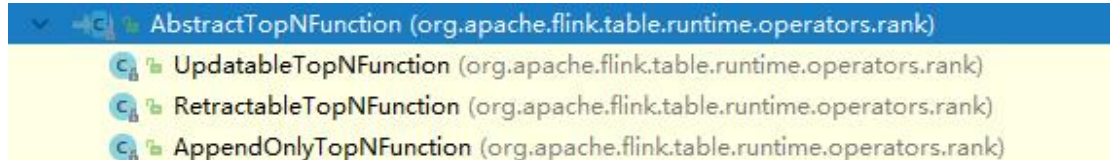
### 7.2.1 使用最优算法

当 TopN 的输出是非更新流（例如 Source），TopN 只有一种算法 AppendRank。当 TopN 的输出是更新流时（例如经过了 AGG/JOIN 计算），TopN 有 2 种算法，性能从高到低分别是：UpdateFastRank 和 RetractRank。算法名字会显示在拓扑图的节点名字上。



注意: apache 社区版的 Flink1.12 目前还没有 UnaryUpdateRank, 阿里云实时计算版 Flink 才有

```
// TODO Use UnaryUpdateTopNFunction after SortedMapState is merged
case RetractStrategy =>
  val equaliserCodeGen = new EqualiserCodeGenerator(inputRowTypeInfo.toRowFieldTypes)
  val generatedEqualiser = equaliserCodeGen.generateRecordEqualiser( name = "RankValueEqualiser")
  val comparator = new ComparableRecordComparator(
    sortKeyComparator,
    sortFields.indices.toArray,
    sortKeyType.toRowFieldTypes,
    sortDirections,
    nullsIsLast)
  new RetractableTopNFunction(
    minIdleStateRetentionTime,
    maxIdleStateRetentionTime,
    inputRowTypeInfo,
    comparator,
    sortKeySelector,
    rankType,
    rankRange,
    generatedEqualiser,
    generateUpdateBefore,
    outputRankNumber)
```



➤ **UpdateFastRank**：最优算法

需要具备 2 个条件：

- 1) 输入流有 **PK (Primary Key)** 信息，例如 Group BY AVG。
- 2) **排序字段的更新是单调的**，且单调方向与排序方向相反。例如，ORDER BY COUNT/COUNT\_DISTINCT/SUM (**正数**) **DESC**。

如果要获取到优化 Plan，则您需要在 **使用 ORDER BY SUM DESC 时**，添加 SUM 为正数的过滤条件。

➤ **AppendFast**：结果只追加，不更新

➤ **RetractRank**：普通算法，性能差

不建议在生产环境使用该算法。请检查输入流是否存在 PK 信息，如果存在，则可进行 UpdateFastRank 优化。

## 7.2.2 无排名优化（解决数据膨胀问题）

### ➤ TopN 语法：

```
SELECT *
FROM (
  SELECT *,
    ROW_NUMBER() OVER ([PARTITION BY col1[, col2..]]
    ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
  FROM table_name)
WHERE rownum <= N [AND conditions]
```

### ➤ 数据膨胀问题：

根据 TopN 的语法，rownum 字段会作为结果表的主键字段之一写入结果表。但是这可能导致数据膨胀的问题。例如，收到一条原排名 9 的更新数据，更新后排名上升到 1，则从 1 到 9 的数据排名都发生变化了，需要将这些数据作为更新都写入结果表。这样就产生了数据膨胀，导致结果表因为收到了太多的数据而降低更新速度。

### ➤ 使用方式

TopN 的输出结果**不需要显示 rownum 值**，仅需在最终前端显式时进行 1 次排序，极大地减少输入结果表的数据量。只需要在外层查询中将 rownum 字段裁剪掉即可

```
// 最外层的字段，不写 rownum
SELECT col1, col2, col3
FROM (
  SELECT col1, col2, col3
    ROW_NUMBER() OVER ([PARTITION BY col1[, col2..]]
    ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
  FROM table_name)
WHERE rownum <= N [AND conditions]
```

在无 rownum 的场景中，对于结果表主键的定义需要特别小心。如果定义有误，会直接导致 TopN 结果的不正确。无 rownum 场景中，主键应为 TopN 上游 GROUP BY 节点的 KEY 列表。

## 7.2.3 增加 TopN 的 Cache 大小

TopN 为了提升性能有一个 State Cache 层，Cache 层能提升对 State 的访问效率。

TopN 的 Cache 命中率的计算公式为。

```
cache_hit = cache_size*parallelism/top_n/partition_key_num
```

例如，Top100 配置缓存 10000 条，并发 50，当 PartitionBy 的 key 维度较大时，例如 10 万级别时，Cache 命中率只有  $10000*50/100/100000=5\%$ ，命中率会很低，导致大量的请求都会击中 State（磁盘），性能会大幅下降。因此当 PartitionKey 维度特别大时，可以适当加大 TopN 的 Cache size，相对应的也建议适当加大 TopN 节点的 Heap Memory。

### ➤ 使用方式

```
// 初始化 table environment
TableEnvironment tEnv = ...

// 获取 tableEnv 的配置对象
Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数:
// 默认 10000 条, 调整 TopN cache 到 20 万, 那么理论命中率能达  $200000*50/100/100000=100\%$ 
configuration.setString("table.exec.topn.cache-size", "200000");
```

注意：目前源码中标记为实验项，官网中未列出该参数



```
object StreamExecRank {
  // It is a experimental config, will may be removed later.
  @Experimental
  val TABLE_EXEC_TOPN_CACHE_SIZE: ConfigOption[Long] =
    key( key = "table.exec.topn.cache-size" )
      .defaultValue(Long.valueOf(10000L))
      .withDescription( description = "TopN operator has a cache which caches partial state contents to reduce" +
        " state access. Cache size is the number of records in each TopN task." )
}
```

It is a experimental config, will may be removed later.  
译文：这是一个实验性的配置，稍后可能会被删除。  
[进入翻译页面 >](#)

## 7.2.4 PartitionBy 的字段中要有时间类字段

例如每天的排名，要带上 Day 字段。否则 TopN 的结果到最后会由于 State ttl 有错乱。

## 7.2.5 优化后的 SQL 示例

```
insert
into print_test
SELECT
  cate_id,
  seller_id,
  stat_date,
  pay_ord_amt  --不输出 rownum 字段，能减小结果表的输出量（无排名优化）
FROM (
  SELECT
```

```

*,
ROW_NUMBER () OVER (
    PARTITION BY cate_id,
    stat_date --注意要有时间字段, 否则 state 过期会导致数据错乱 (分区字段优化)
    ORDER
        BY pay_ord_amt DESC --根据上游 sum 结果排序。排序字段的更新是单调的, 且单调方向与排序方向相反 (走最优算法)
    ) as rownum
FROM (
    SELECT
        cate_id,
        seller_id,
        stat_date,
        --重点。声明 Sum 的参数都是正数, 所以 Sum 的结果是单调递增的, 因此 TopN 能使用
        优化算法, 只获取前 100 个数据 (走最优算法)
        sum (total_fee) filter (
            where
                total_fee >= 0
        ) as pay_ord_amt
    FROM
        random_test
    WHERE
        total_fee >= 0
    GROUP
        BY cate_name,
        seller_id,
        stat_date
    ) a
WHERE
    rownum <= 100
);

```

## 7.3 高效去重方案

由于 SQL 上没有直接支持去重的语法, 还要灵活的保留第一条或保留最后一条。因此我们使用了 SQL 的 ROW\_NUMBER OVER WINDOW 功能来实现去重语法。去重本质上是一种特殊的 TopN。

### 7.3.1 保留首行的去重策略 (Deduplicate Keep FirstRow)

保留 KEY 下第一条出现的数据, 之后出现该 KEY 下的数据会被丢弃掉。因为 STATE 中只存储了 KEY 数据, 所以性能较优, 示例如下:

```

SELECT *
FROM (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY b ORDER BY proctime) as rowNum
    FROM T

```

```
)  
WHERE rowNum = 1;
```

以上示例是将 T 表按照 b 字段进行去重，并按照系统时间保留第一条数据。Proctime 在这里是源表 T 中的一个具有 Processing Time 属性的字段。如果按照系统时间去重，也可以将 Proctime 字段简化 PROCTIME()函数调用，可以省略 Proctime 字段的声明。

## 7.3.2 保留末行的去重策略 (Deduplicate Keep LastRow)

保留 KEY 下最后一条出现的数据。保留末行的去重策略性能略优于 **LAST\_VALUE** 函数，

示例如下：

```
SELECT *  
FROM (  
  SELECT *,  
    ROW_NUMBER() OVER (PARTITION BY b, d ORDER BY rowtime DESC) as rowNum  
  FROM T  
)  
WHERE rowNum = 1;
```

以上示例是将 T 表按照 b 和 d 字段进行去重，并按照业务时间保留最后一条数据。

Rowtime 在这里是源表 T 中的一个具有 Event Time 属性的字段。

## 7.4 高效的内置函数

### 7.4.1 使用内置函数替换自定义函数

Flink 的内置函数在持续的优化当中，请尽量使用内部函数替换自定义函数。使用内置函数好处：

- 1) 优化数据序列化和反序列化的耗时。
- 2) 新增直接对字节单位进行操作的功能。

支持的系统内置函数：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/functions/systemFunctions.html>



## 7.4.2 LIKE 操作注意事项

- 如果需要进行 StartWith 操作, 使用 LIKE 'xxx%'。
- 如果需要进行 EndWith 操作, 使用 LIKE '%xxx'。
- 如果需要进行 Contains 操作, 使用 LIKE '%xxx%'。
- 如果需要进行 Equals 操作, 使用 LIKE 'xxx', 等价于 str = 'xxx'。
- 如果需要匹配 \_ 字符, 请注意要完成转义 LIKE '%seller/id%' ESCAPE '/'。\_ 在 SQL 中属于单字符通配符, 能匹配任何字符。如果声明为 LIKE '%seller\_id%', 则不单会匹配 seller\_id 还会匹配 seller#id、sellerxid 或 seller1id 等, 导致结果错误。

## 7.4.3 慎用正则函数 (REGEXP)

正则表达式是非常耗时的操作, 对比加减乘除通常有百倍的性能开销, 而且正则表达式在某些极端情况下可能会进入无限循环, 导致作业阻塞。建议使用 LIKE。正则函数包括:

- REGEXP
- REGEXP\_EXTRACT
- REGEXP\_REPLACE

## 7.5 指定时区

本地时区定义了当前会话时区 id。当本地时区的时间戳进行转换时使用。在内部, 带有本地时区的时间戳总是以 **UTC 时区** 表示。但是, 当转换为不包含时区的数据类型时(例如 TIMESTAMP, TIME 或简单的 STRING), 会话时区在转换期间被使用。为了避免时区错乱的问题, 可以参数指定时区。

```
// 初始化 table environment  
TableEnvironment tEnv = ...
```

```
// 获取 tableEnv 的配置对象
Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数:
// 指定时区
configuration.setString("table.local-time-zone", "Asia/Shanghai");
```

## 7.6 设置参数总结

总结以上的调优参数，代码如下：

```
// 初始化 table environment
TableEnvironment tEnv = ...

// 获取 tableEnv 的配置对象
Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数:
// 开启 miniBatch
configuration.setString("table.exec.mini-batch.enabled", "true");
// 批量输出的间隔时间
configuration.setString("table.exec.mini-batch.allow-latency", "5 s");
// 防止 OOM 设置每个批次最多缓存数据的条数，可以设为 2 万条
configuration.setString("table.exec.mini-batch.size", "20000");
// 开启 LocalGlobal
configuration.setString("table.optimizer.agg-phase-strategy",
    "TWO_PHASE");
// 开启 Split Distinct
configuration.setString("table.optimizer.distinct-agg.split.enabled",
    "true");
// 第一层打散的 bucket 数目
configuration.setString("table.optimizer.distinct-agg.split.bucket-num",
    "1024");
// TopN 的缓存条数
configuration.setString("table.exec.topn.cache-size", "200000");
// 指定时区
configuration.setString("table.local-time-zone", "Asia/Shanghai");
```