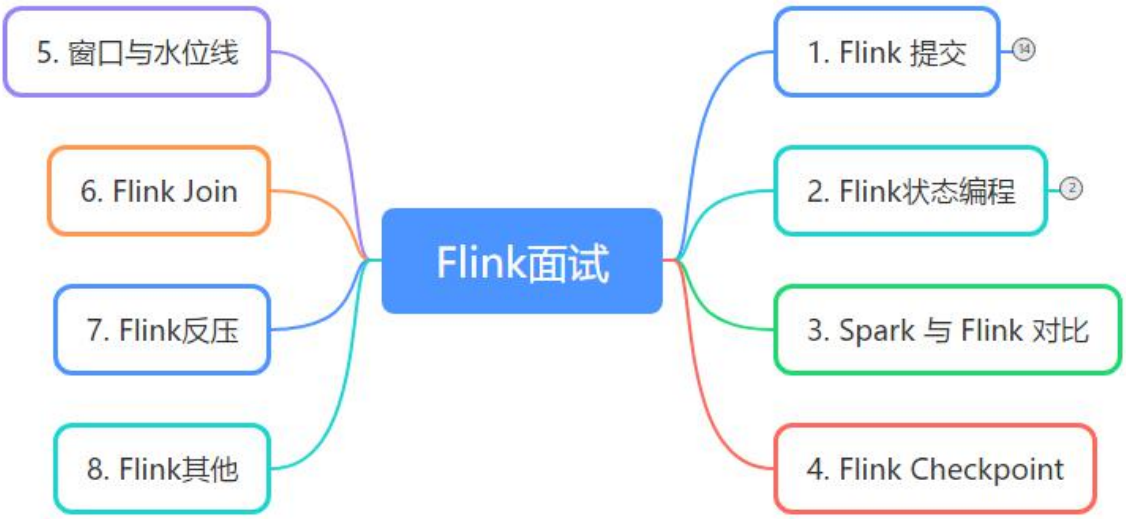
Flink 8 大主题，50 个知识点，2 万字助力面试准备

零.前言

主要内容如下



关注下方公众号回复关键字【资料】即可获取。欢迎大家指正错误与不足之处，也欢迎大家继续补充此内容。

新增的内容会重新放到资料中，大家可回复关键字【进群】,有资料修改更新或增添新内容会首发群中通知。

一. Flink 提交系列

1. Flink 怎么提交?

Local 模式

JobManager 和 TaskManager 共用一个 JVM,只需要 jdk 支持，单节点运行，主要用来调试。

Standlone 模式

Standlone 是 Flink 自带的一个分布式集群，它不依赖其他的资源调度框架、不依赖 yarn 等。充当 Master 角色的是 JobManager。充当 Slave/Worker 角色是 TaskManager

Yarn 模式

Yarn 模式	生命周期	资源隔离	优点	缺点	main 方法
Session	关闭会话,才会停止	共用 JM 和 TM	预先启动，启动作业不再启动。资源充分共享	资源隔离比较差，TM 不容易扩展	在客户端执行
Per-job	Job 停止，集群停止	单个 Job 独享 JM 和 TM	充分隔离，资源根据 job 按需申请	job 启动慢,每个 job 需要启动一个 JobManager	在客户端执行
Application	当 Application 全部执行	Application 使用一套 JM 和 TM	Client 负载低，Application	对 per-job 模式和 session 模式	在 Cluster

Yarn 模式	生命周期	资源隔离	优点	缺点	main 方法
	完，集群才会停止		之间实现资源隔离，Application 内实现资源共享	的优化部署模式(优点)	

2. Flink 集群规模？在 Flink 项目做了什么？

Flink 群集大小时要考虑的一些方面:

1.记录数和每条记录的大小 每秒到达流式传输框架的预期记录数以及每条记录的大小。不同的记录类型将具有不同的大小，这将最终影响 Flink 应用程序平稳运行所需的资源。

2.不同 key 的数量和每个键的状态大小。

3.状态更新的数量和状态后端的访问模式 Java 的堆状态后端上的各种访问模式可能会显着影响群集的大小以及 Flink 作业所需的资源。

4.网络容量 网络容量不仅会受到 Flink 应用程序本身的影响，还会受到可能正在与之交互的外部服务（如 Kafka 或 HDFS）的影响。此类外部服务可能会导致网络出现额外流量。例如，启用 replication 可能会在网络的消息 brokers 之间创建额外的流量。

5.磁盘带宽。

6.机器数量及其可用 CPU 和内存。

Flink 项目做了什么？

实时监控：

1. 用户行为预警，服务器攻击预警 .....

实时报表：

1. 活动直播大屏: 双 11、双 12
2. 对外数据产品时效性
3. 数据化运营

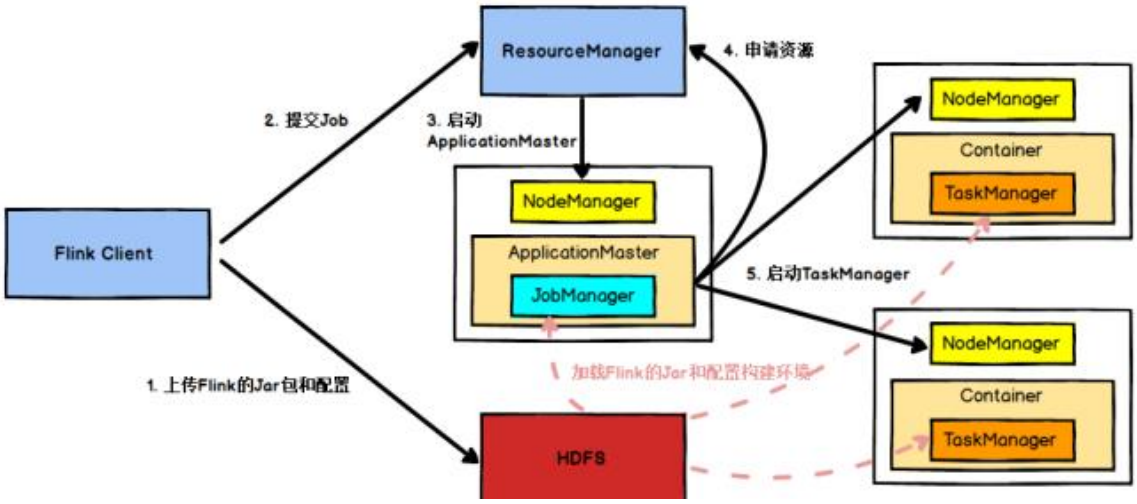
流数据分析：

1. 实时计算相关指标反馈及时调整决策
2. 内容投放、无线智能推送、实时个性化推荐等

实时仓库：

1. 数据实时清洗、归并、结构化
2. 数仓的补充和优化

3. Flink 提交作业的流程，以及与 Yarn 怎么交互?



（1）提交 App 之前，先上传 Flink 的 Jar 包和配置到 HDFS，以便 JobManager 和 TaskManager 共享 HDFS 的数据。

（2）客户端向 ResourceManager 提交 Job，ResouceManager 接到请求后，先分配 container 资源，然后通知 NodeManager 启动 ApplicationMaster。

（3）ApplicationMaster 会加载 HDFS 的配置，启动对应的 JobManager，然后 JobManager 会分析当前的作业图，将它转化成执行

图（包含了所有可以并发执行的任务），从而知道当前需要的具体资源。

（4）接着，JobManager 会向 ResourceManager 申请资源，ResouceManager 接到请求后，继续分配 container 资源，然后通知 ApplictaionMaster 启动更多的 TaskManager（先分配好 container 资源，再启动 TaskManager）。container 在启动 TaskManager 时也会从 HDFS 加载数据。

（5）TaskManager 启动后，会向 JobManager 发送心跳包。JobManager 向 TaskManager 分配任务。

4. Flink 提交 Job 的方式以及参数设置？

```
./bin/flink run -t yarn-session \

  -Dyarn.application.id=application_XXXX_YY \

  ./examples/streaming/TopSpeedWindowing.jar



 ./bin/flink run -t yarn-per-job

  --detached ./examples/streaming/TopSpeedWindowing.jar



 ./bin/flink run-application -t yarn-application

 ./examples/streaming/TopSpeedWindowing.jar



yn(实际) = Math.ceil(p/ys)

ys(总共) = yn(实际) * ys(指定)

ys(使用) = p(指定)
```

```
flink run

-c,--class Flink 应用程序的入口

-C,--classpath 指定所有节点都可以访问到的 url,可用于多个应用程序都需要的工具类加载

-d,--detached 是否使用分离模式，就是提交任务,cli 是否退出,加了-d 参数,cli 会退出

-n,--allowNonRestoredState 允许跳过无法还原的 savepoint。比如删除了代码中的部分 operator

-p,--parallelism 执行并行度

-s,--fromSavepoint 从 savepoint 恢复任务

-sae,--shutdownOnAttachedExit 以 attached 模式提交，客户端退出的时候关闭集群

flink yarn-cluster 模式

-d,--detached 是否使用分离模式

-m,--jobmanager 指定提交的 jobmanager
```

```
-yat,--yarnapplicationType 设置 yarn 应用的类型

-yD 使用给定属性的值

-yd,--yarndetached 使用 yarn 分离模式

-yh,--yarnhelp yarn session 的帮助

-yid,--yarnapplicationId 挂到正在运行的 yarnsession 上

-yj,--yarnjar Flink jar 文件的路径

-yjm,--yarnjobManagerMemory jobmanager 的内存(单位 M)

-ynl,--yarnnodeLabel 指定 YARN 应用程序 YARN 节点标签

-ynm,--yarnname 自定义 yarn 应用名称

-yq,--yarnquery 显示 yarn 的可用资源

-yqu,--yarnqueue 指定 yarn 队列

-ys,--yarnslots 指定每个 taskmanager 的 slots 数

-yt,--yarnship 在指定目录中传输文件

-ytm,--yarntaskManagerMemory 每个 taskmanager 的内存
```

```
-yz,--yarnzookeeperNamespace 用来创建 ha 的 zk 子路径的命名空间

-z,--zookeeperNamespace 用来创建 ha 的 zk 子路径的命名空间
```

5. Flink 的 JobManger? 有多少个 JobManager?

JobManger

（1） 控制一个应用程序执行的主进程，也就是说，每个应用程序 都会被一个不同的 JM 所控制执行。

（2） JM 会先接收到要执行的应用程序，这个应用程序会包括：作业图（Job Graph）、逻辑数据流图（ ogical dataflow graph）和打包了所有的类、库和其它资源的 JAR 包。

（3） JM 会把 Jobgraph 转换成一个物理层面的 数据流图，这个图被叫做 “执行图”(Executiongraph),包含了所有可以并发执行的任务。Job Manager 会向资源管理器（Resourcemanager)请求执行任务必要的资源，也就是 任务管理器(Taskmanager)上的插槽 slot。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 TM 上。而在运行过程中 JM 会负责所有需要中央协调的操作，比如说检查点(checkpoints)的协调。

有多少个 JobManager

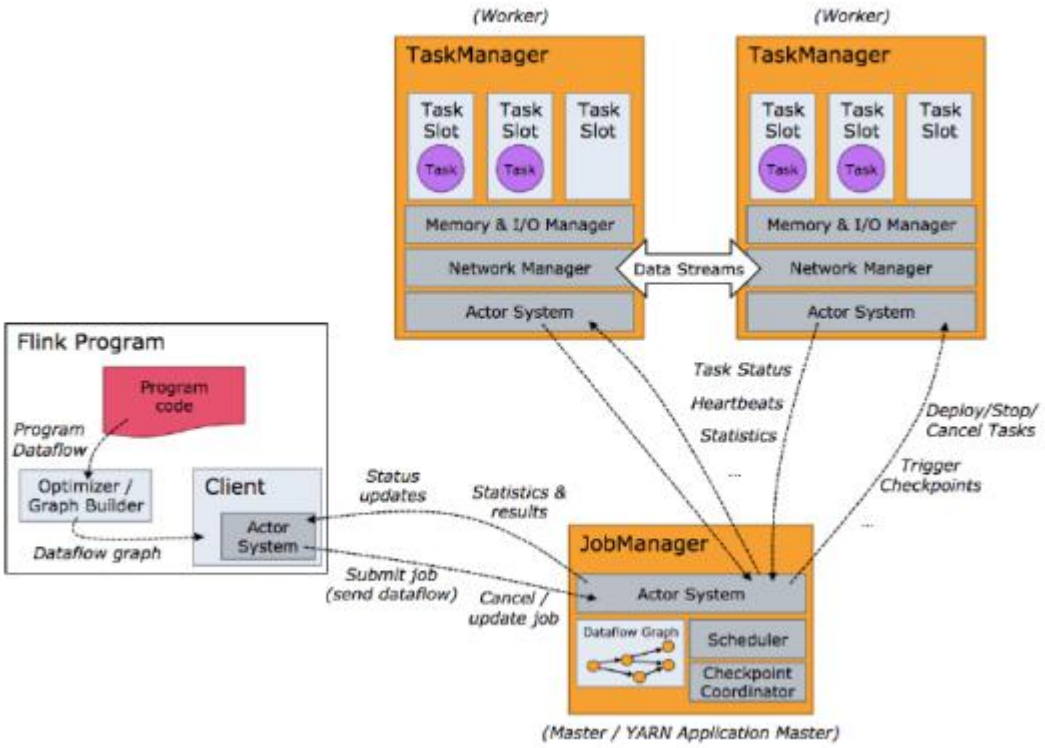
集群默认只有一个 Job Manager。但为了防止单点故障，我们配置了高可用。我们公司一般配置一个主 JobManager，两个备用 JobManager，然后结合 ZooKeeper 的使用，来达到高可用。

6. JobManger 在集群启动过程中起到什么作用？

JobManager 的职责主要是接收 Flink 作业，调度 Task，收集作业状态和管理 TaskManager。它包含一个 Actor，并且做如下操作：

RegisterTaskManager: 它由想要注册到 JobManager 的 TaskManager 发送。注册成功会通过 AcknowledgeRegistration 消息进行 Ack。
SubmitJob: 由提交作业到系统的 Client 发送。提交的信息是 JobGraph 形式的作业描述信息。
CancelJob: 请求取消指定 id 的作业。成功会返回 CancellationSuccess ，否则返回 CancellationFailure。
UpdateTaskExecutionState: 由 TaskManager 发送，用来更新执行节点 (ExecutionVertex) 的状态。成功则返回 true，否则返回 false。
RequestNextInputSplit: TaskManager 上的 Task 请求下一个输入 split ，成功则返回 NextInputSplit，否则返回 null。
JobStatusChanged：它意味着作业的状态(RUNNING, CANCELING, FINISHED,等)发生变化。这个消息由 ExecutionGraph 发送。

7. Flink 的 TaskManager?



（1） Flink 中的工作进程。通常在 Flink 中会有多个 TM 运行， 每个 TM 都包含了一定数量的插槽 slots。插槽的数量限制了 TM 能够执行的任务数量。

（2） 启动之后，TM 会向资源管理器注册它的插槽；收到资源管理器的指令后， TM 就会将一个或者多个插槽提供给 JM 调用。TM 就可以向插槽分配任务 tasks 来执行了。

（3） 在执行过程中， 一个 TM 可以跟其它运行同一应用程序的 TM 交换数据。

TaskManager 相当于整个集群的 Slave 节点，负责具体的任务执行和对应任务在每个 节点上的资源申请和管理。客户端通过将编写好的 Flink 应用编译打包，提交到 JobManager，然后 JobManager 会根据已注册在 JobManager 中 TaskManager 的资源情况，将任务分配给有资源的 TaskManager 节点，然后启动并运行任务。TaskManager 从 JobManager 接收需要部署的任务，然后使用 Slot 资源启动 Task，建立数据接入的网络连接，接收数据并开始数据处理。同时 TaskManager 之间的数据交互都是通过数据流的方式进行的。可以看出，Flink 的任务运行其实是采用多线程的方式，这和 MapReduce 多 JVM 进行的方式有很大的区别，Flink 能够极大提高 CPU 使用效率，在多个任务和 Task 之间通过 TaskSlot 方式共享系统资源，每个 TaskManager 中通过管理多个 TaskSlot 资源池进 行对资源进行有效管理。

8. 说一下 slot，业务中一个 TaskManager 设置几个 slot？

jobManager：负责接收 Flink Client 提交的 Job，并将 Job 分发到 TaskManager 执行，一个 JobManager 包含一个或多个 TaskManager。

TaskManager：负责执行 Client 提交的 Job。每个 TaskManager 可以有一个或多个 slot，但 slot 的个数不能多于 cpu-cores。

slot：slot 是 Flink 任务的最小执行单位，并行度上限不能大于 slot 的数量。

9. Flink 的并行度？

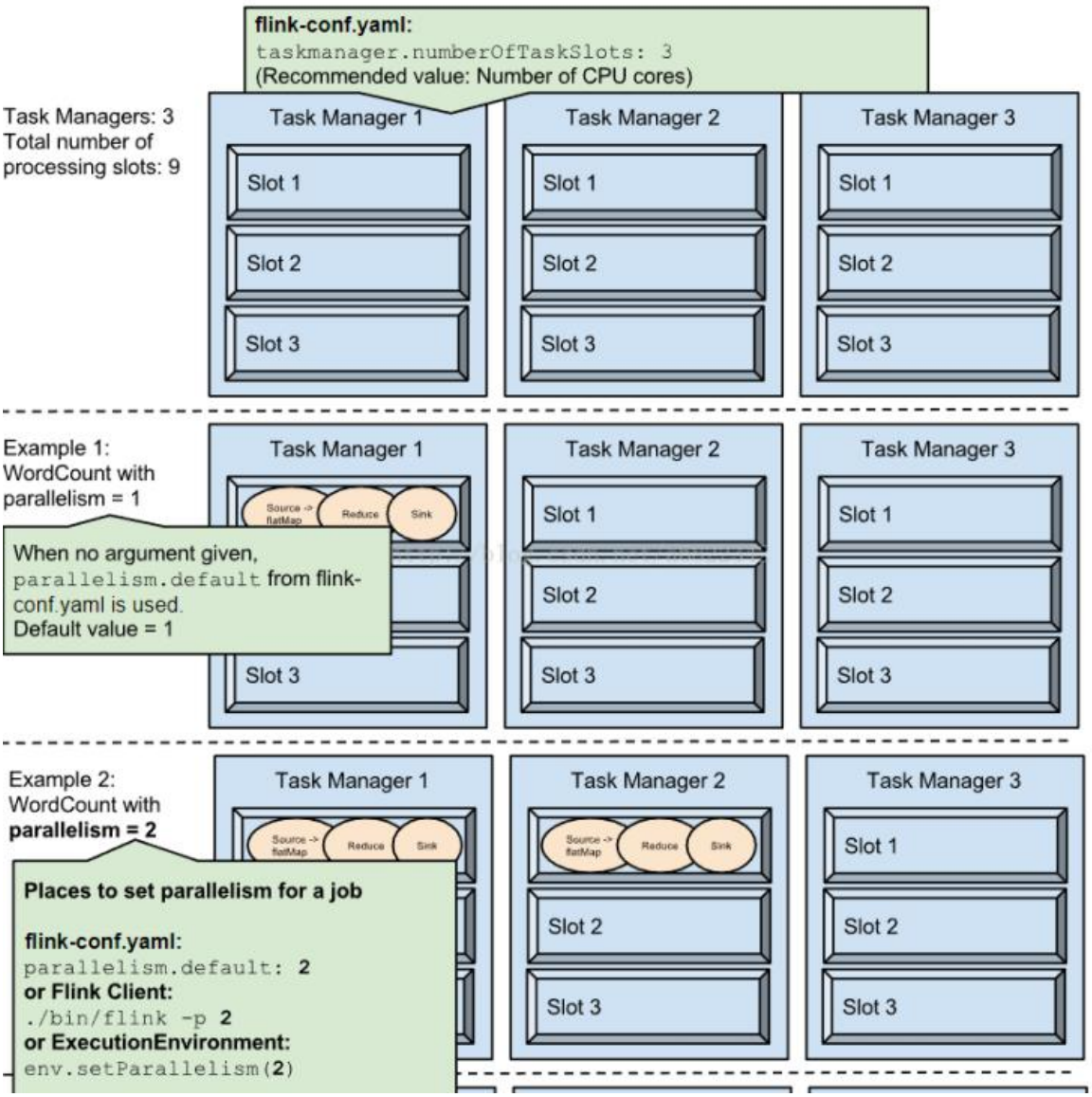
Flink 中的任务被分为多个并行任务来执行，其中每个并行的实例处理一部分数据。这些并行实例的数量被称为并行度。设置并行度一般在四个层面设置（优先级由高到低）
操作算子层面
执行环境层面
客户端层面
系统层面

10. Flink 计算资源的调度是如何实现的？

TaskManager 中最细粒度的资源是 Taskslot，代表了一个固定大小的资源子集，每个 TaskManager 会将其所占有的资源平分给它的 slot。
通过调整 task slot 的数量，用户可以定义 task 之间是如何相互隔离的。每个 TaskManager 有一个 slot，也就意味着每个 task 运行在独立的 JVM 中。每个 TaskManager 有多个 slot 的话，也就是说多个 task 运行在同一个 JVM 中。而在同一个 JVM 进程中的 task，可以共享 TCP 连接（基于多路复用）和心跳消息，可以减少数据的网络传输，也能共享一些数据结构，一定程度上减少了每个 task 的消耗。每 个 slot 可以接受单个 task，也可以接受多个连续 task 组成的 pipeline。

11. Flink 的 Slot 和 Parallelism

slot 是指 taskmanager 的并发执行能力，假设我们将 taskmanager.numberOfTaskSlots 配置为 3 那么每一个 taskmanager 中分配 3 个 TaskSlot,3 个 taskmanager 一共有 9 个 TaskSlot。
parallelism 是指 taskmanager 实际使用的并发能力。假设我们把 parallelism.default 设置为 1，那么 9 个 TaskSlot 只能用 1 个，有 8 个空闲。



12. Operator Chains 了解吗？

为了更高效地分布式执行，Flink 会尽可能地将 operator 的 subtask 链接（chain）在一起形成 task。每个 task 在一个线程中执行。

将 operators 链接成 task 是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量。这就是 Operator Chains（算子链）。

13. Flink 的运行必须依赖 Hadoop 组件吗？

Flink 可以完全独立于 Hadoop，在不依赖 Hadoop 组件下运行。
但是 Flink 集成 Yarn 做资源调度，也可以读写 HDFS，或者利用 HDFS 做检查点。

14. 怎么修改正在运行的 Flink 程序？如果有新的实时指标你们是怎么上线的？

正在运行的 Flink 程序：修改不了。可动态加载配置广播等等
完成作业开发
作业调试，并且通过语法检查后
上线作业，即可将数据发布至生产环境。

二. 状态编程系列

1. 说一下状态编程（operator state，keyed state）

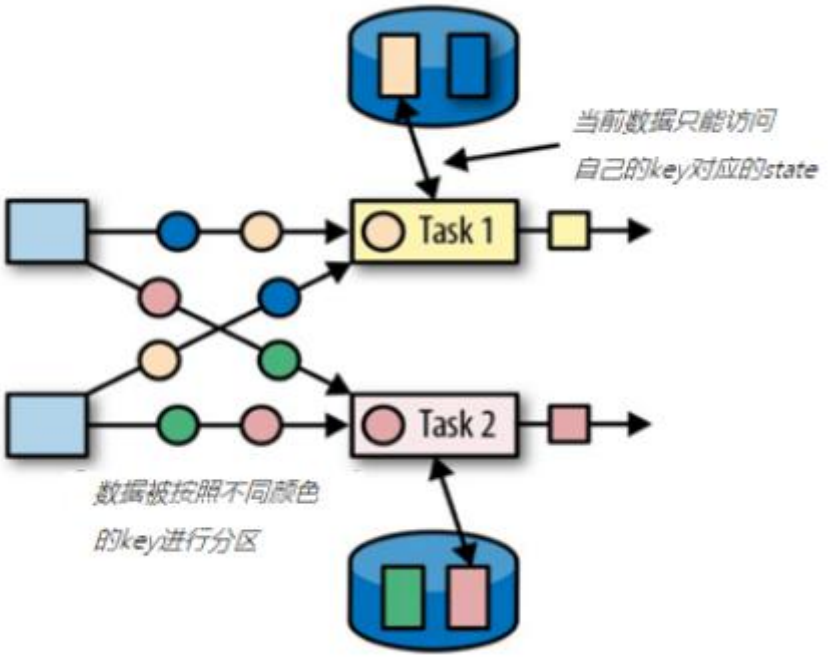
我们知道 Flink 上的聚合和窗口操作，一般都是基于 KeyedStream 的， 数据会按照 key 的哈希值进行分区，聚合处理的结果也应该是只对当前 key 有效。然而同一个分区（也就是 slot）上执行的任务实例，可能会包含多个 key 的数据， 它们同时访问和更改本地变量，就会导致计算结果错误。所以这时状态并不是单纯的本地变量。
容错性，也就是故障后的恢复。状态只保存在内存中显然是不够稳定的，我们需要将它持久化保存， 做一个备份；在发生故障后可以从这个备份中恢复状态。
处理的数据量增大时，我们应该相应地对计算资源扩容， 调大并行度。这时就涉及到了状态的重组调整。
状态的分类：

1. 托管状态（Managed State）和原始状态（Raw State）

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。
对于具体的状态内容，Flink 也提供了值状态（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。
聚合、窗口等算子中内置的状态，就都是托管状态。
我们也可以在富函数类（RichFunction）中通过上下文来自定义状态，这些也都是托管状态。

算子状态（Operator State）和按键分区状态

1. 按键分区状态
按键分区状态（Keyed State）顾名思义，是任务按照键（key）来访问和维护的状态。它的特点非常鲜明，就是以 key 为作用范围进行隔离。Keyed State 类似于一个分布式的映射（map）数据结构，所有的状态会根据 key 保存成键值对（key-value）的形式。这样当一条数据到来时，任务就会自动将状态的访问范围限定为当前数据的 key，从 map 存储中读取出对应的状态值。所以具有相同 key 的所有数据都会到访问相同的状态，而不同 key 的状态之间是彼此隔离的。



值状态（ValueState）：

状态中只保存一个“值”（value）。

列表状态（ListState）：

以列表（List）的形式组织起来。在 ListState 接口中同样有一个类型参数 T，表示列表中数据的类型。ListState 也提供了一系列的方法来操作状态，使用方式与一般的 List 非常相似。

映射状态（MapState）：

把一些键值对（key-value）作为状态整体保存起来，可以认为就是一组 key-value 映射的列表。

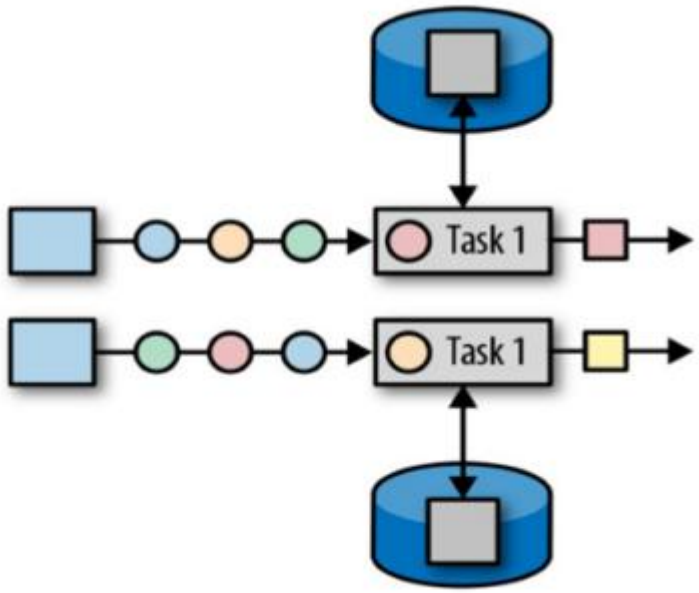
归约状态（ReducingState）：

类似于值状态（Value），不过需要对添加进来的所有数据进行归约，将归约聚合之后的值作为状态保存下来。

聚合状态（AggregatingState）：

与归约状态非常类似，聚合状态也是一个值，用来保存添加进来的所有数据的聚合结果。

2. 算子状态



算子状态（Operator State）就是一个算子并行实例上定义的状态，作用范围被限定为当前算子任务。算子状态跟数据的 key 无关，所以不同 key 的数据只要被分发到同一个并行子任务， 就会访问到同一个 Operator State。

算子状态的实际应用场景不如 Keyed State 多，一般用在 Source 或 Sink 等与外部系统连接的算子上， 或者完全没有 key 定义的场景。比如 Flink 的 Kafka 连接器中，就用到了算子状态。

当算子的并行度发生变化时，算子状态也支持在并行的算子任务实例之间做重组分配。根据状态的类型不同，重组分配的方案也会不同。

状态类型:ListState、UnionListState 和 BroadcastState。

ListState

Keyed State 中的列表状态的区别是：在算子状态的上下文中，不会按键（key）分别处理状态， 所以每一个并行子任务上只会保留一个“列表”（list），也就是当前并行子任务上所有状态项的集合。

列表中的状态项就是可以重新分配的最细粒度，彼此之间完全独立。当算子并行度进行缩放调整时，算子的列表状态中的所有元素项会被统一收集起来，相当于把多个分区的列表合并成了一个“大列表”， 然后再均匀地分配给所有并行任务。这种“均匀分配”的具体方法就是“轮询”（round-robin），与之前介绍的 rebanlance 数据传输方式类似， 是通

过逐一“发牌”的方式将状态项平均分配的。这种方式也叫作“平均分割重组”（even-split redistribution）。

算子状态中不会存在“键组”（key group）这样的结构，所以为了方便重组分配，就把它直接定义成了“列表”（list）。这也就解释了，为什么算子状态中没有最简单的值状态（ValueState）。

联合列表状态（UnionListState）

与 ListState 类似，联合列表状态也会将状态表示为一个列表。它与常规列表状态的区别在于，算子并行度进行缩放调整时对于状态的分配方式不同。UnionListState 的重点就在于“联合”（union）。在并行度调整时，常规列表状态是轮询分配状态项，而联合列表状态的算子则会直接广播状态的完整列表。这样，并行度缩放之后的并行子任务就获取到了联合后完整的“大列表”，可以自行选择要使用的状态项和要丢弃的状态项。这种分配也叫作“联合重组”（union redistribution）。如果列表中状态项数量太多，为资源和效率考虑一般不建议使用联合重组的方式。

广播状态（BroadcastState）

有时我们希望算子并行子任务都保持同一份“全局”状态，用来做统一的配置和规则设定。

这时所有分区的所有数据都会访问到同一个状态，状态就像被“广播”到所有分区一样，这种特殊的算子状态，就叫作广播状态（BroadcastState）。因为广播状态在每个并行子任务上的实例都一样，所以在并行度调整的时候就比较简单，只要复制一份到新的并行任务就可以实现扩展；而对于并行度缩小的情况，可以将多余的并行子任务连同状态直接砍掉——因为状态都是复制出来的，并不会丢失。

2. 10 个 int 以数组的形式保存,保存在什么状态好?VlaueState 还是 ListState?存在哪个的性能比较好?

ValueState[Array[Int]] update 形式。

ListState[Int]: add 形式添加。

对于操控来说 ListState 方便取值与更改。
按键分区状态（Keyed State）选择 ValueState ListState。
算子状态（Operator State）选择 ListState。

3. 使用 MapStage，group by id 如何设计？id 不放在 key 行不行？

MapState 是 KeyedState，也就是 keyBy 后才能使用 MapState。所以 State 中肯定要保存 key。
group by id。假设 id 有 id1、id2 这两个值，id1、id2 就是 key。
id 的设计是结合业务场景，就是把同类数据或者同逻辑数据放到一起计算处理。

4. Flink 是如何管理 kafka 的 offset，使用什么类型的状态保存 offset？

checkpoint 是 Flink 的内部机制，可以从故障中恢复。通俗的理解是 checkpoint 是 Flink 应用程序状态的一致性副本，包括输入的读取位置(offset)。如果发生故障，Flink 将通过从 checkpoint 加载状态后端并从恢复的读取位置继续恢复应用程序，可以做到所谓的断点续传。
checkpoint 使 Flink 具有容错能力，并确保在发生故障时具有容错的能力。应用程序可以定期触发检查点。
Flink 中的 Kafka 消费者将 Flink 的检查点机制与有状态运算符集成在一起， 其状态是所有 Kafka 分区中的读取偏移量。触发 checkpoint 时， 每个分区的偏移量都存储在 checkpoint 中。Flink 的 checkpoint 机制确保所有操作员任务的存储状态是一致的。

即它们基于相同的输入数据。当所有操作员任务成功存储其状态时，检查点完成。因此，当从潜在的系统故障重新启动时，系统提供一次性状态更新保证。

```
private transient ListState<Tuple2<KafkaTopicPartition, Long>> unio
nOffsetStates;



public final void snapshotState(FunctionSnapshotContext context) th
rows Exception {
        if (!running) {
            LOG.debug("snapshotState() called on closed source");
        } else {
            unionOffsetStates.clear();

            final AbstractFetcher<?, ?> fetcher = this.kafkaFetcher;
            if (fetcher == null) {
                // the fetcher has not yet been initialized, which
means we need to return the
                // originally restored offsets or the assigned part
itions
                for (Map.Entry<KafkaTopicPartition, Long> subscribe
dPartition :
                        subscribedPartitionsToStartOffsets.entrySet
()) {
                    unionOffsetStates.add(
                            Tuple2.of(
```

```
                                    subscribedPartition.getKey(), s
ubscribedPartition.getValue()));
                }

                if (offsetCommitMode == OffsetCommitMode.ON_CHECKPO
INTS) {
                    // the map cannot be asynchronously updated, be
cause only one checkpoint call
                    // can happen
                    // on this function at a time: either snapshotS
tate() or
                    // notifyCheckpointComplete()
                    pendingOffsetsToCommit.put(context.getCheckpoin
tId(), restoredState);
                }
            } else {
                HashMap<KafkaTopicPartition, Long> currentOffsets =
 fetcher.snapshotCurrentState();

                if (offsetCommitMode == OffsetCommitMode.ON_CHECKPO
INTS) {
                    // the map cannot be asynchronously updated, be
cause only one checkpoint call
                    // can happen
                    // on this function at a time: either snapshotS
tate() or
                    // notifyCheckpointComplete()
```

```
                    pendingOffsetsToCommit.put(context.getCheckpoin
tId(), currentOffsets);
                }

                for (Map.Entry<KafkaTopicPartition, Long> kafkaTopi
cPartitionLongEntry :
                        currentOffsets.entrySet()) {
                    unionOffsetStates.add(
                            Tuple2.of(
                                    kafkaTopicPartitionLongEntry.ge
tKey(),
                                    kafkaTopicPartitionLongEntry.ge
tValue()));
                }
            }

            if (offsetCommitMode == OffsetCommitMode.ON_CHECKPOINTS)
 {
                // truncate the map of pending offsets to commit, t
o prevent infinite growth
                while (pendingOffsetsToCommit.size() > MAX_NUM_PEND
ING_CHECKPOINTS) {
                    pendingOffsetsToCommit.remove(0);
                }
            }
        }
    }
```

5. 一个窗口，现在只取第一帧和最后一帧，怎么做？

如果需要访问窗口中的最后一个元素， 则可能应该只使用 WindowFunction 并在 apply 方法中获取可迭代输入（input.last）的第一个和最后一个元素。
getstart getend 获取时间。

三. 反压与问题系列

1. Flink 用什么监控，监控什么？如何有效处理数据积压？

我们监控了 Flink 的任务是否停止，的 Kafka 的 LAG，我们会进行实时数据对账

1. Flink Web UI

比如反压：通过 Thread.getStackTrace() 采集在 TaskManager 上正在运行的所有线程， 收集在缓冲区请求中阻塞的线程数（意味着下游阻塞），并计算缓冲区阻塞线程数与总线程数的比值 rate。其中，rate < 0.1 为 OK，0.1 <= rate <= 0.5 为 LOW，rate > 0.5 为 HIGH。

2. Prometheus&Grafana 监控

作业的可用性，如 uptime (作业持续运行的时间)、fullRestarts (作业重启的次数)

作业的流量，可以通过 numRecordsIn、numBytesInLocal 等相关指标来关注作业每天处理的消息数目及高峰时间段的流量， 通过关注这些指标可以观察作业的流量表现是否正常。

CPU（如：CPU.Load）、内存（如：Heap.Used ）、GC (如：GarbageCollector.Count、GarbageCollector.Time)及网络 (inputQueueLength、outputQueueLength) 相关指标，这些指标一般是用来排查作业的故障信息。

checkpoint 相关信息，例如最近完成的 checkpoint 的时长（ lastCheckpointDuration ）、 最近完成的 checkpoint 的大小

（ lastCheckpointSize ）、 作业失败后恢复的能力
（ lastCheckpointRestoreTimestamp ）、 成功和失败的 checkpoint 数目（ numberOfCompletedCheckpoints、
numberOfFailedCheckpoints ） 以及在 Exactly once 模式下 barrier 对齐时间（ checkpointAlignmentTime ）
connector 的指标，例如常用的 Kafka connector ，Kafka 自身提供了一些指标，可以帮助我们了解到作业最新消费的消息的状况、 作业是否有延迟等
其他自定义指标：超时丢弃的数据量， filter 过滤的数据量/占比 处理失败的数据，等等

背压

背压产生的原因：下游消费的速度跟不上上游生产数据的速度，可能出现的原因如下：
（1）节点有性能瓶颈，可能是该节点所在的机器有网络、磁盘等等故障，机器的网络延迟和磁盘不足、频繁 GC、数据热点等原因。
（2）数据源生产数据的速度过快，计算框架处理不及时。比如消息中间件 kafka，生产者生产数据过快，下游 flink 消费计算不及时。
（3）flink 算子间并行度不同，下游算子相比上游算子过小。
背压导致的影响 首先，背压不会直接导致系统的崩盘，只是处在一个不健康的运行状态。
（1）背压会导致流处理作业数据延迟的增加。
（2）影响到 Checkpoint，导致失败，导致状态数据保存不了，如果上游是 kafka 数据源，在一致性的要求下，可能会导致 offset 的提交不上。
原理: 由于 Flink 的 Checkpoint 机制需要进行 Barrier 对齐，如果此时某个 Task 出现了背压，Barrier 流动的速度就会变慢，导致 Checkpoint 整体时间变长，如果背压很严重，还有可能导致 Checkpoint 超时失败。

（3）影响 state 的大小，还是因为 checkpoint barrier 对齐要求。导致 state 变大。

原理：接受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 state 变大。

Flink 不需要一个特殊的机制来处理背压，因为 Flink 中的数据传输相当于已经提供了应对背压的机制。所以只有从代码上与资源上去做一些调整。

（1）背压部分原因可能是由于数据倾斜造成的，我们可以通过 Web UI 各个 SubTask 的 指标值来确认。Checkpoint detail 里不同 SubTask 的 State size 也是一个分析数据倾斜的有用指标。解决方式把数据分组的 key 预聚合来消除数据倾斜。

（2）代码的执行效率问题，阻塞或者性能问题。

（3）TaskManager 的内存大小导致背压。

2. 遇到 Flink 不太能解决的问题？（PV,UV 放内存，OOM 了，后面配合 redis 以及布隆过滤器）

有时候需要求 uv ，内存或者状态中存过多数据，导致压力巨大， 这个时候可以结合 Redis 或者 布隆过滤器来去重。

注意：布隆过滤器存在非常小的误判几率，不能判断某个元素一定百分之百存在，所以只能用在允许有少量误判的场景，不能用在需要 100% 精确判断存在的场景。

3. 使用 flink 统计订单表的 GMV，如果 mysql 中的数据出现错误，之后在 mysql 中做数据的修改操作，那么 flink 程序如何保证 GMV 的正确性，你们是如何解决？

一般有离线 Job 来恢复和完善实时数据。

四. Spark 与 Flink 对比

1. Spark 与 Flink 区别

1. 架构模型：Spark Streaming 在运行时的主要角色包括：Master、Worker、Driver、Executor，Flink 在运行时主要包含：Jobmanager、Taskmanager 和 Slot。
2. 任务调度：Spark Streaming 连续不断的生成微小的数据批次，构建有向无环图 DAG，Spark Streaming 会依次创建 DStreamGraph、JobGenerator、JobScheduler。Flink 根据用户提交的代码生成 StreamGraph，经过优化生成 JobGraph，然后提交给 JobManager 进行处理，JobManager 会根据 JobGraph 生成 ExecutionGraph，ExecutionGraph 是 Flink 调度最核心的数据结构，JobManager 根据 ExecutionGraph 对 Job 进行调度。
3. 时间机制：Spark Streaming 支持的时间机制有限，只支持处理时间。Flink 支持了流处理程序在时间上的三个定义：处理时间、事件时间、注入时间。同时也支持 watermark 机制来处理滞后数据。
4. 容错机制：对于 Spark Streaming 任务，我们可以设置 checkpoint，然后假如发生故障并重启，我们可以从上次 checkpoint 之处恢复，但是这个行为只能使得数据不丢失，可能会重复处理，不能做到恰好一次处理语义。Flink 则使用两阶段提交协议来解决这个问题。

2. Flink 的 key By 和 Spark 的 group by 有什么区别?

keyBy 算子将 DataStream 转换成一个 KeyedStream。KeyedStream 是一种特殊的 DataStream， 事实上，KeyedStream 继承了

DataStream，DataStream 的各元素随机分布在各 Task Slot 中，KeyedStream 的各元素按照 Key 分组，分配到各 Task Slot 中。groupBy 按照传入函数的返回值进行分组。将相同的 key 对应的值放入一个迭代器。

3. 为什么要用 Flink 替代 SparkStreaming?

Spark 和 Flink 都具有流和批处理能力，但是他们的做法是相反的。Spark Streaming 是把流转化成一个个小的批来处理，这种方案的一个问题是我们需要的延迟越低， 额外开销占的比例就会越大，这导致了 Spark Streaming 很难做到秒级甚至亚秒级的延迟。Flink 是把批当作一种有限的流，这种做法的一个特点是在流和批共享大部分代码的同时还能够保留批处理特有的一系列的优化。
同时，Flink 相比于 Spark 而言还有诸多明显优势：
支持高效容错的状态管理，保证在任何时间都能计算出正确的结果；同时支持高吞吐、低延迟、高性能的分布式流式数据处理框架；支持事件时间（Event Time）概念，事件即使无序到达甚至延迟到达，数据流都能够计算出精确的结果；轻量级分布式快照（Snapshot）实现的容错，能将计算过程分布到单台并行节点上进行处理。

五. Checkpoint 系列

1. Flink checkpoint 的实现原理

Flink 在新版本上改造之前方式：暂停应用, 然后开始做检查点, 再重新恢复应用 这种方式效率低，所以 Flink 改造分布式快照算，异步 barrier 快照。
每个需要 checkpoint 的应用在启动时，Flink 的 JobManager 为其创建一个 CheckpointCoordinator， CheckpointCoordinator 全权负责本应用的快照制作。

如果接着问 Barrier 是啥？多个 barrier 被插入到数据流中, 然后作为数据流的一部分随着数据流动(有点类似于 Watermark).这些 barrier 不会跨越流中的数据. 每个 barrier 会把数据流分成两部分: 一部分数据进入当前的快照 , 另一部分数据进入下一个快照 . 每个 barrier 携带着快照的 id. barrier 不会暂停数据的流动, 所以非常轻量级. 在流中, 同一时间可以有来源于多个不同快照的多个 barrier, 这个意味着可以并发的出现不同的快照。

Flink 的检查点制作过程

1. Checkpoint Coordinator 向所有 source 节点 trigger Checkpoint. 然后 Source Task 会在数据流中安插 CheckPoint barrier

Job Manager 对每一个 job 都会产生一个 Checkpoint Coordinator 向所有 source 节点 触发 trigger Checkpoint 节点， 并行度是几，就会触发多少个。source 会向流中触发 Barrier，接收到 Barrier 的节点就会保存快照（包括 source）。

2. source 节点向下游广播 barrier，这个 barrier 就是实现 Chandy-Lamport 分布式快照算法的核心， 下游的 task 只有收到所有进来的 barrier 才会执行相应的 Checkpoint(barrier 对齐） 注意：新版本有一种新非对齐 barrier

source 先收到 barrier，然后往后传递，若是多并行度，相当于多组接力赛跑比赛，所以顺序是不一致的，并不是同步。

3. 下游的 sink 节点收集齐上游两个 input 的 barrier 之后，会执行本地快照。
4. 同样的，sink 节点在完成自己的 Checkpoint 之后，会将 state handle 返回通知 Coordinator。
5. 最后，当 Checkpoint coordinator 收集齐所有 task 的 state handle，就认为这一次的 Checkpoint 全局完成了，向持久化存储中再备份一个 Checkpoint meta 文件。

2. Checkpoint 存储

MemoryStateBackend

内存级的状态后端，会将键控状态作为内存中的对象进行管理，将它们存储 在 TaskManager 的 JVM 堆上；而将 checkpoint 存储在 JobManager 的内存中。

FsStateBackend

将 checkpoint 存到远程的持久化文件系统（FileSystem）上。而对于本地状态， 跟 MemoryStateBackend 一样，也会存在 TaskManager 的 JVM 堆上。

RocksDBStateBackend

将所有状态序列化后，存入本地的 RocksDB 中存储。

追问：了解 RocksDB 与 RocksDBStateBackend？

RocksDB 是一个用于快速存储的可嵌入持久化键值存储。它通过 Java Native 接口（JNI）与 Flink 进行交互。Flink 作业运行时，RocksDB 会被内嵌到 TaskManager 进程中。

RocksDBStateBackend 还支持作为性能调优选项的增量 checkpoint。增量 checkpoint 仅存储上次 checkpoint 之后发生的改变。与执行完整快照相比，这大大减少了 checkpoint 的时间。

场景

1. 作业的状态大小大于地内存(如较长的窗口，较大的 Keyed 状态)。
2. 作业需要使用增量 checkpoint，以减少 checkpoint 的时间。
3. 作业需要保证可预测的延迟，不受 JVM 垃圾回收的影响。

4. Flink 的 checkpoint 机制以及精准一次性消费如何实现？

1. souce：使用执行 ExactlyOnce 的数据源，比如 kafka 等
2. 内部使用 FlinkKafakConsumer，并开启 CheckPoint，偏移量会保存到 StateBackend 中，并且默认会将偏移量写入到 topic 中去，即 _consumer_offsets Flink 设置 CheckepointingModel.EXACTLY_ONCE

3. sink：存储系统支持覆盖也即幂等性：如 Redis,Hbase,ES 等 存储系统不支持覆：需要支持事务(预写式日志或者两阶段提交),两阶段提交可参考 Flink 集成的 kafka sink 的实现

3. 精确一次，至少一次对 checkpoint 有什么影响？

精确一次: 在多并行度下, 如果要实现严格一次, 则要执行 barrier 对齐.
当 job graph 中的每个 operator 接收到 barriers 时，它就会记录下其状态。有且只有一次，消息不丢失不重复，且只消费一次
至少一次语义: barrier 不对齐,会重复消费。如果不对齐，那么在 chk-100 快照之前，已经处理了一些 chk-100 对应的 offset 之后的数据，当程序从 chk-100 恢复任务时，chk-100 对应的 offset 之后的数据还会被处理一次， 所以就出现了重复消费。

4. Savapoint 了解多少？

savepoint 是“通过 checkpoint 机制”创建的，所以 savepoint 本质上是特殊的 checkpoint
savepoint 的侧重点是“维护”，即 Flink 作业需要在人工干预下手动重启、升级、迁移或 A/B 测试时，先将状态整体写入可靠存储，维护完毕之后再从 savepoint 恢复现场
savepoint 则以二进制形式存储所有状态数据和元数据
avepoint 并不会连续自动触发，所以 savepoint 没有必要支持增量

5. Flink checkpoint 的超时问题 如何解决?

1、是否网络问题
2、是否是 barrir 问题
3、查看 webui，是否有数据倾斜

4、有数据倾斜的话，那么解决数据倾斜后，会有改善，

6. 作业挂掉了，恢复上一个 Checkpoint，用什么命令？

```
-s hdfs://192.168.0.1:8020/flink/checkpoint/...
```

7. 什么是 Flink 的非 barrier 对齐，如何实现？

非对齐总体流程：在接受上游多个输入情况下，每一个批次的 checkpoint 不会发生数据缓存， 会直接交给下游去处理，checkpoint 信息会被缓存在一个 CheckpointBarrierCount 类型的队列中，CheckpointBarrierCount 标识了一次 checkpoint 与其 channel 输入 checkpointBarrier 个数， 当 checkpointBarrier 个数与 channel 个数相同则会触发 checkpoint。

六. 窗口 与 Watermark 系列

1. Flink 时间语义

事件时间 Event Time
处理时间 Process Time
进入时间 Ingestion Time

2. 什么是 Watermark 及主要作用？什么时候去触发计算?

Watermark 是一种衡量 Event Time 进展的机制，用于处理乱序事件的，单调递增的时间戳；数据流中的 Watermark 用于表示 timestamp 小于 Watermark 的数据，都已经到达；
watermark(水位线，包含延迟) > 窗口结束时间

3. 消息超过 watermark 的时间会丢失数据吗?

allowedLateness 也是 Flink 处理乱序事件的一个特别重要的特性,默认情况下, 当 wartermark 通过 window 后,再进来的数据,也就是迟到或者晚到的数据就会别丢弃掉了, 但是有的时候我们希望在一个可以接受的范围内, 迟到的数据,也可以被处理或者计算,这就是 allowedLateness 产生的原因了

迟到的元素也可以使用侧输出(side output)特性被重定向到另外的一条流中去。迟到元素所组成的侧输出流可以继续处理或者 sink 到持久化设施中去。

4. 开窗函数有哪些?

```
FlinkSQL

窗口:

TUMBLE(TABLE data, DESCRIPTOR(timecol), size)


HOP(TABLE data, DESCRIPTOR(timecol), slide, size [, offset ])


SESSION(<time-attr>, <gap-interval>)

<gap-interval>: INTERVAL 'string' timeUnit


累积窗口函数: CUMULATE(TABLE data, DESCRIPTOR(timecol), step, size)


窗口分组聚合 GROUPING SETS


over 函数
```

```
CUBE



Flink DataStream

增量聚合和全量聚合

增量聚合：窗口不维护原始数据，只维护中间结果，每次基于中间结果和增量数据进行

聚合。

如：ReduceFunction、AggregateFunction


全量聚合：窗口需要维护全部原始数据，窗口触发进行全量聚合。

如:ProcessWindowFunction
```

5. (没有数据流的时候)窗口,这个窗口存在吗?

没有数据，窗口不产生

6. 1 小时的滚动窗口,一小时处理一次的压力比较大,想让他 5 分钟处理一次.怎么办?

自定义触发器，4 个方法，一个 Close 三个用于控制计算和输出

七. Join 系列

1. Flink 的双流 join 的底层原理?

union：union 支持双流 Join，也支持多流 Join。多个流类型必须一致；
connector：connector 支持双流 Join，两个流的类型可以不一致；

join：该方法只支持 inner join，即：相同窗口下，两个流中，Key 都存在且相同时才会关联成功；
coGroup：同样能够实现双流 Join。即：将同一 Window 窗口内的两个 DataStream 联合起来，两个流按照 Key 来进行关联，并通过 apply()方法 new CoGroupFunction() 的形式，重写 join() 方法进行逻辑处理。
intervalJoin：Interval Join 没有 Window 窗口的概念，直接用时间戳作为关联的条件，更具表达力。

基于 Connect 的双流 JOIN 实现机制

对两个 DataStream 执行 connect 操作，将其转化为 ConnectedStreams, 生成的 Streams 可以调用不同方法在两个实时流上执行，且双流之间可以共享状态。
orderStream.connect(orderDetailStream) .keyBy("orderId", "orderId") .process(new orderProcessFunc());
总体思想：基于数据时间实现订单数据及订单明细数据的关联，超时或者缺失则由侧输出流输出。
在 connect 中针对订单流和订单明细流，先创建定时器并保存 state 状态，处于窗口内就进行 join, 否则进入侧输出流。

基于 Window Join 的双流 JOIN 实现机制

将两条实时流中元素分配到同一个时间窗口中完成 Join。底层原理: 两条实时流数据缓存在 Window State 中，当窗口触发计算时，执行 join 操作。

```
orderStream.join(orderDetailStream)
    .where(r => r._1)  //订单 id
    .equalTo(r => r._2) //订单 id
```

```
    .window(TumblingProcessTimeWindows.of(
          Time.seconds(60)))
    .apply {(r1, r2) => r1 + " : " + r2}
```

coGroup 算子

coGroup 算子也是基于 window 窗口机制，不过 coGroup 算子比 Join 算子更加灵活，可以按照用户指定的逻辑匹配左流或右流数据并输出。

```
orderDetailStream
  .coGroup(orderStream)
  .where(r -> r.getOrderId())
  .equalTo(r -> r.getOrderId())
  .window(TumblingProcessingTimeWindows.of(Time.seconds(60)))
  .apply(new CoGroupFunction<OrderDetail, Order, Tuple2<String, Lon
g>>() {
    @Override
    public void coGroup(Iterable<OrderDetail> orderDetailRecords, I
terable<Order> orderRecords, Collector<Tuple2<String, Long>> collec
tor)  {
      for (OrderDetail orderDetaill : orderDetailRecords) {
        boolean flag = false;
        for (Order orderRecord : orderRecords) {
          // 右流中有对应的记录
          collector.collect(new Tuple2<>(orderDetailRecords.getGood
s_name(), orderDetailRecords.getGoods_price()));
          flag = true;
```

```
        }
        if (!flag) {
          // 右流中没有对应的记录
          collector.collect(new Tuple2<>(orderDetailRecords.getGood
s_name(), null));
        }
      }
    }
  })
  .print();
```

2. A 表 left join B 表

(1）A 表数据来了，B 没来
(2）A 表数据来了，B 在规定时间内到
(3)A 表数据来了，B 在规定时间后面到
怎么处理？

```
left join 与 right join 由于 Flink 官方并没有给出明确的方案，无法通过 join
来实现，但是可以用 coGroup


public static class LeftJoin implements CoGroupFunction<Tuple3<Stri
ng, String, Long>, Tuple3<String, String,
            Long>, Tuple5<String,
            String, String, Long, Long>> {


        // 将 key 相同，并且在同一窗口的数据取出来
        @Override
        public void coGroup(Iterable<Tuple3<String, String, Long>>
```

```
first, Iterable<Tuple3<String, String, Long>> second,
                            Collector<Tuple5<String, String, String,
 Long, Long>> out) throws Exception {

            for (Tuple3<String, String, Long> leftElem : first) {
                boolean hadElements = false;

                //如果左边的流 join 上了右边的流 rightElements 就不为空，
就会走下面的增强 for 循环
                for (Tuple3<String, String, Long> rightElem : secon
d) {
                    //将 join 上的数据输出
                    out.collect(new Tuple5<>(leftElem.f0, leftElem.
f1, rightElem.f1, leftElem.f2,
                            rightElem.f2));
                    hadElements = true;
                }

                if (!hadElements) {
                    //没 join 上，给右边的数据赋空值
                    out.collect(new Tuple5<>(leftElem.f0, leftElem.
f1, "null", leftElem.f2, -1L));
                }
            }
        }
    }
```

```
    对于正常数据，直接 join

    对于其他情况，看业务需求吧，不需要立马舍弃的，可用状态或者第三方存储来等
待
```

3. Flink 维表关联怎么做的？

1、async io
2、broadcast
3、async io + cache
4、open 方法中读取，然后定时线程刷新，缓存更新是先删除，之后再来一条之后再负责写入缓存

八. 其他

1. process 用的种类？

ProcessFunction 用来构建事件驱动的应用以及实现自定义的业务逻辑(使用之前的 window 函数和转换算子无法实现)。例如，Flink SQL 就是使用 Process Function 实现的。
ProcessFunction 是一个低级的流处理操作，允许访问所有(非循环)流应用程序的基本构件：
events：数据流中的元素 state：状态，用于容错和一致性，仅用于 keyed stream timers：定时器，支持事件时间和处理时间，仅用于 keyed stream Flink 提供了 8 个 Process Function：
ProcessFunction：dataStream KeyedProcessFunction：用于 KeyedStream，keyBy 之后的流处理
CoProcessFunction：用于 connect 连接的流

ProcessJoinFunction：用于 join 流操作

BroadcastProcessFunction：用于广播

KeyedBroadcastProcessFunction：keyBy 之后的广播

ProcessWindowFunction：窗口增量聚合

ProcessAllWindowFunction：全窗口聚合

可以将 ProcessFunction 看作是一个具有 key state 和定时器(timer)访问权的 FlatMapFunction。对于在输入流中接收到的每一个事件，此函数就会被调用以处理该事件。

对于容错状态，ProcessFunction 可以通过 RuntimeContext 访问 Flink 的 keyed state，这与其他有状态函数访问 keyed state 的方式类似。
定时器可让应用程序对在处理时间和事件时间中的变化进行响应。每次调用 processElement(...)函数时都可以获得一个 Context 对象，通过该对象可以访问元素的事件时间（event time）时间戳以及 TimerService。可以使用 TimerService 为将来的事件时间/处理时间实例注册回调。对于事件时间计时器，当当前水印被提升到或超过计时器的时间戳时，将调用 onTimer(...)方法，而对于处理时间计时器，当挂钟时间达到指定时间时，将调用 onTimer(...)方法。在调用期间，所有状态的范围再次限定为创建定时器所用的 key，从而允许定时器操作 keyed state。
如果想要在流处理过程中访问 keyed state 和定时器，就必须在一个 keyed stream 上应用 ProcessFunction 函数，代码如下：

```
stream.keyBy(...).process(new MyProcessFunction())
```

2. Flink 的内存管理？

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上。此外，Flink 大量的使用了堆外内存。如果需要处理的数据超出了内存限制，则会将部分数据 存储到硬盘上。Flink 为了直接操作二进制数据实现了自己的序列化框架。理论上 Flink 的。内存管理分为三部分：

Network Buffers

这个是在 TaskManager 启动的时候分配的，这是一组用于缓存网络数据的内存，每个块是 32K，默认分配 2048 个，可以通过“taskmanager.network.numberOfBuffers” 修改。

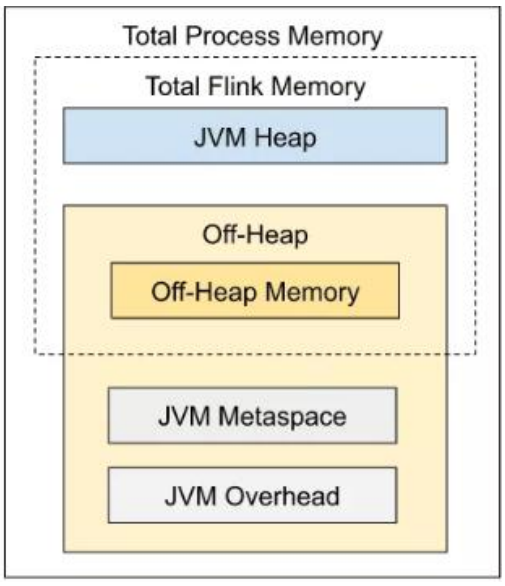
Memory Manage pool

大量的 Memory Segment 块，用于运行时的算法（Sort/Join/Shuffle 等），这部分启动的时候就会分配。

User Code

这部分是除了 Memory Manager 之外的内存用于 User code 和 TaskManager 本身的数据结构。

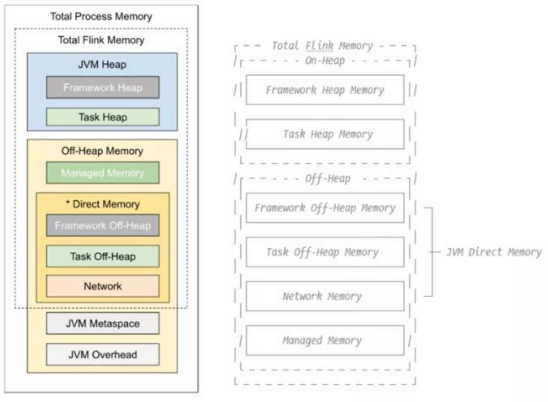
JobManager 内存模型



配置 JobManager 的总进程内存

```
#The heap size for the JobManager JVM


jobmanager.heap.size:1024m


#Flink1.11 版本及以后


#JobManager 总进程内存

jobmanager.memory.process.size:4096m


# 作业管理器的 JVM 堆内存大小

jobmanager.memory.heap.size: 2048m


# 作业管理器的堆外内存大小。此选项涵盖所有堆外内存使用。

jobmanager.memory.off-heap.size: 1536m
```

TaskManager 内存模型



TaskManager 内存模型一共包含 3 大部分，分别为总体内存、JVM Heap 堆上内存、Off-Heap 堆外内存等。

总体内存

1. Total Process Memory：Flink Java 应用程序（包括用户代码）和 JVM 运行整个进程所消耗的总内存。

总进程内存(Total Process Memory) = Flink 总内存 + JVM 元空间 + JVM 执行开销。

2. Total Flink Memory：仅 Flink Java 应用程序消耗的内存，包括用户代码，但不包括 JVM 为其运行而分配的内存。

Flink 总内存 = Framework 堆内外 + task 堆内外 + network + managed Memory

JVM Heap (JVM 堆上内存)

1. Framework Heap ：框架堆内存
2. Task Heap : 任务堆内存 如果内存大小没有指定，它将被推导出为总 Flink 内存减去框架堆内存、框架堆外内存、任务堆外内存、托管内存和网络内存。

Off-Heap Mempry(JVM 堆外内存)

1. Managed memory: 托管内存

由 Flink 管理的原生托管内存，保留用于排序、哈希表、中间结果缓存和 RocksDB 状态后端。

2. DirectMemory：JVM 直接内存

Framework Off-Heap Memory：Flink 框架堆外内存。
Task Off-Heap ： Task 堆外内存。
Network Memory：网络内存。

3. JVM metaspace：JVM 元空间。默认为 256MB。
4. JVM Overhead ： JVM 执行开销

3. Flink 的序列化机制？

当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个 Java 对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为 Java 对象。把 Java 对象转换为字节序列的过程称为对象的序列化。
把字节序列恢复为 Java 对象的过程称为对象的反序列化。
Apache Flink 摒弃了 Java 原生的序列化方法，以独特的方式处理 数据类型和序列化，包含自己的类型描述符，泛型类型提取和类型序列化框架。TypeInformation 是所有类型描述符的基类。它揭示了该类型的一些基本属性，并且可以生成序列化器。TypeInformation 支持以下几种类型：
BasicTypeInfo: 任意 Java 基本类型或 String 类型
BasicArrayTypeInfo: 任意 Java 基本类型数组或 String 数组
WritableTypeInfo: 任意 Hadoop Writable 接口的实现类
TupleTypeInfo: 任意的 Flink Tuple 类型。
CaseClassTypeInfo: 任意的 Scala CaseClass。
PojoTypeInfo: 任意的 POJO (Java or Scala)，例如，Java 对象的所有成员变量，要么是 public 修饰符定义，要么有 getter/setter 方法。
GenericTypeInfo: 任意无法匹配之前几种类型的类
针对前六种类型数据集，Flink 皆可以自动生成对应的 TypeSerializer，能非常高效地对数据集进行序列化和反序列化。

4. Rich Functions 与 Functions 区别?

Rich 版本。它与常规函数的不同在于，可以获取运行环境的上下文，并拥有一些生命周期方法， 所以可以实现更复杂的功能。也有意味着提供了更多的，更丰富的功能。默认生命周期方法：1.初始化方法 open(), 在每个并行度上只会被调用一次, 而且先被调用；
2.最后一个方法 close(), 做一些清理工作, 在每个并行度上只调用一次, 而且是最后被调用， 但读文件时在每个并行度上调用两次；
3.getRuntimeContext()方法提供了函数的 RuntimeContext 的一些信息， 可以通过改方法获取到函数执行并行度，任务名称，state 状态等信息

5. Flink 分区策略?

策略	描述
Global	记录发送给下游 operator 的第一个实例
shuffle	记录随机发送给下游 operator 的每一个实例
rebalance	记录循环发送给下游 operator 的每一个实例
forward	记录输出到下游的 operator 实例。要求上下游算子并行度一致。上下游算子同属一个 task
hash	按 key 的 hash 值输出到下游 operator 实例

策略	描述
rescale	基于上游的并行度，将数据以循环的方式输出到下游每个实例
broadcast	广播分区将上游数据输出到下游算子的每个实例中，适合大数据 join 小数据集
custom	自定义方式将记录输出到下游

6. Flink 中的分布式缓存

Flink 读取文件，把数据放在 taskmanager 节点中，防止 task 重复拉取。

```
env.registerCachedFile("hdfs:///", "hdfsFile")
```

7. Flink 里面异步 IO 代码具体怎么写的?

Flink source 收到一条数据就会进行处理，如果需要通过这条数据关联外部数据源， 例如 mysql，在发出查询请求后，同步 IO 的方式是会等待查询结果再处理下一条数据的查询， 也就是每一条数据都要等待上一个查询结束。而异步 IO 是指数据来了以后发出查询请求， 先不等查询结果，直接继续发送下一条的查询请求，对于查询结果是异步返回的， 返回结果之后再进入下一个算子的计算。

1. 定义一个 AsyncFunction 用于实现请求的分发
2. 定义一个 callback 回调函数，该函数用于取出异步请求的返回结果，并将返回的结果传递给 ResultFuture .
3. 对 DataStream 使用 Async 操作。

```
class AsyncDatabaseRequest extends AsyncFunction[String, (String, S
tring)] {

    /** 可以异步请求的特定数据库的客户端 */
    lazy val client: DatabaseClient = new DatabaseClient(host, post,
 credentials)

    /** future 的回调的执行上下文（当前线程） */
    implicit lazy val executor: ExecutionContext = ExecutionContext.
fromExecutor(Executors.directExecutor())

    override def asyncInvoke(str: String, asyncCollector: AsyncColl
ector[(String, String)]): Unit = {

        // 发起一个异步请求，返回结果的 future
        val resultFuture: Future[String] = client.query(str)

        // 设置请求完成时的回调: 将结果传递给 collector
        resultFuture.onSuccess {
            case result: String => asyncCollector.collect(Iterable(
(str, result)));
        }
    }
}
// 创建一个原始的流
val stream: DataStream[String] = ...
```

```
// 添加一个 async I/O
val resultStream: DataStream[(String, String)] =
    AsyncDataStream.(un)orderedWait(
      stream, new AsyncDatabaseRequest(),
      500, TimeUnit.MILLISECONDS, // 超时时间
      120)  // 进行中的异步请求的最大数量
```

Capacity

该参数用于定义同时最多会有多少个异步请求在同时处理。该参数可以限制并发请求数量，这样的话不会积压非常的未处理请求。

Timeout

可以设置 Timeout 参数来释放挂掉或者失败的请求所占用的资源， 提高资源利用率

超时处理

默认情况下， 当一个异步请求多次超时时，程序会抛出一个异常并重启 job 。但是一般来说我们会重写 AsyncFunction 中的 timeout 方法来自定义超时之后的处理方式.

结果的顺序

AsyncDataStream 包含两种输出模式: 有序和无序，分别对应静态方法 orderedWait 与 unorderedWait。

8. Flink 的重启策略

固定延迟重启策略

固定延迟重启策略是尝试给定次数重新启动作业。如果超过最大尝试次数，则作业失败。在两次连续重启尝试之间，会有一个固定的延迟等待时间。

故障率重启策略

故障率重启策略在故障后重新作业，当设置的故障率（failure rate）超过每个时间间隔的故障时，作业最终失败。在两次连续重启尝试之间，重启策略延迟等待一段时间。

无重启策略

作业直接失败，不尝试重启。

后备重启策略

使用群集定义的重新启动策略。这对于启用检查点的流式传输程序很有帮助。默认情况下，如果没有定义其他重启策略，则选择固定延迟重启策略。

9. Flink 数据倾斜

Flink 任务出现数据倾斜的直观表现是任务节点频繁出现反压。部分节点出现 OOM 异常，是因为大量的数据集中在某个节点上，导致该节点内存被爆，任务失败重启。

方案

数据源 source 消费不均匀

通过调整 Flink 并行度，解决数据源消费不均匀或者数据源反压的情况。我们常常例如 kafka 数据源，调整并行度的原则：Source 并行度与 kafka 分区数是一样的，或者 kafka 分区数是 KafkaSource 并发度的整数倍。建议是并行度等于分区数。

key 分布不均匀

上游数据分布不均匀，使用 keyBy 来打散数据的时候出现倾斜。通过添加随机前缀，打散 key 的分布，使得数据不会集中在几个 Subtask。

两阶段聚合解决 KeyBy（加盐局部聚合+去盐全局聚合）

预聚合：加盐局部聚合，在原来的 key 上加随机的前缀或者后缀。

聚合：去盐全局聚合，删除预聚合添加的前缀或者后缀，然后进行聚合统计。