1. Flink 基础

1.1 简单介绍一下 Flink

Flink 是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。并且 Flink 提供了数据分布、容错机制以及资源管理等核心功能。Flink 提供了诸多高抽象层的 API 以便用户编写分布式任务：

DataSet API， 对静态数据进行批处理操作，将静态数据抽象成分布式的数据集，用户可以方便地使用 Flink 提供的各种操作符对分布式数据集进行处理，支持 Java、Scala 和 Python。

DataStream API，对数据流进行流处理操作，将流式的数据抽象成分布式的数据流，用户可以方便地对分布式数据流进行各种操作，支持 Java 和 Scala。

Table API，对结构化数据进行查询操作，将结构化数据抽象成关系表，并通过类 SQL 的 DSL 对关系表进行各种查询操作，支持 Java 和 Scala。

此外，Flink 还针对特定的应用领域提供了领域库，例如： Flink ML，Flink 的机器学习库，提供了机器学习 Pipelines API 并实现了多种机器学习算法。 Gelly，Flink 的图计算库，提供了图计算的相关 API 及多种图计算算法实现。根据官网的介绍，Flink 的特性包含：

1.2 Flink 相比传统的 Spark Streaming 区别?

这个问题是一个非常宏观的问题，因为两个框架的不同点非常之多。但是在面试时有非常重要的一点一定要回答出来：**Flink 是标准的实时处理引擎，基于事件驱动。而 Spark Streaming 是微批（Micro-Batch）的模型。**

下面我们就分几个方面介绍两个框架的主要区别：

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。

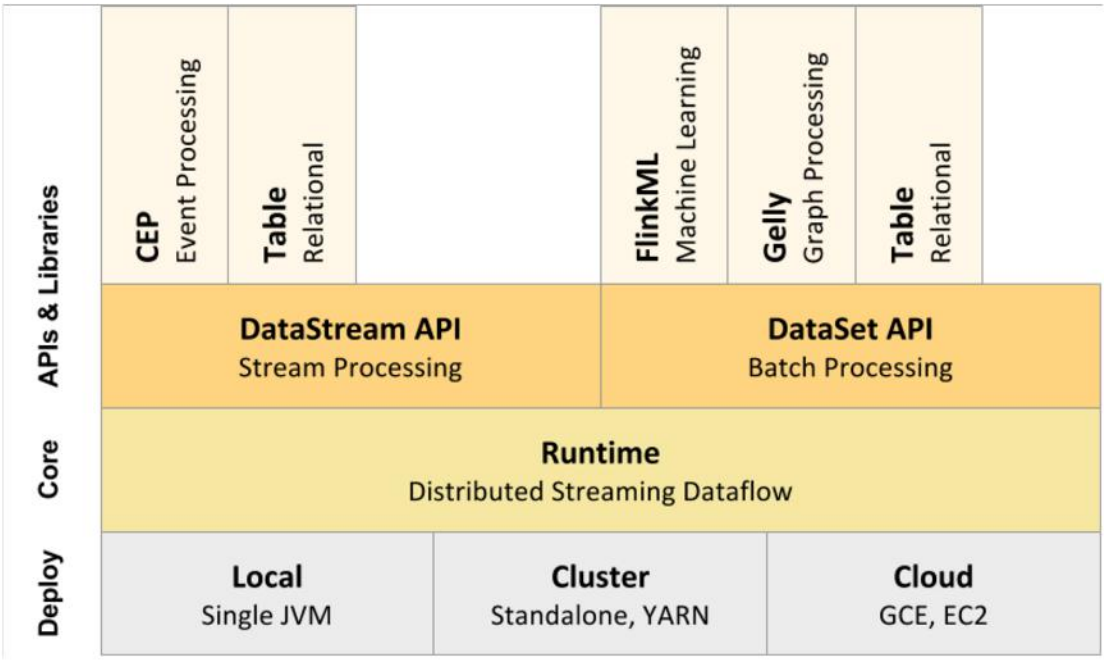
3. 时间机制 Spark Streaming 支持的时间机制有限，只支持处理时间。 Flink 支持了流处理程序在时间上的三个定义：处理时间、事件时间、注入时间。同时也支持 watermark 机

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。

1.3 Flink 的组件栈有哪些？

根据 Flink 官网描述，Flink 是一个分层架构的系统，每一层所包含的组件都提供了特定的抽象，用来服务于上层组件。



自下而上，每一层分别代表：Deploy 层：该层主要涉及了 Flink 的部署模式，在上图中我们可以看出，Flink 支持包括 local、Standalone、Cluster、Cloud 等多种部署模式。Runtime 层：Runtime 层提供了支持 Flink 计算的核心实现，比如：支持分布式 Stream 处理、JobGraph 到 ExecutionGraph 的映射、调度等等，为上层 API 层提供基础服务。API 层：API 层主要实现了面向流（Stream）处理和批（Batch）处理 API，其中面向流处理对应 DataStream API，面向批处理对应 DataSet API，后续版本，Flink 有计划将 DataStream 和 DataSet API 进行统一。Libraries 层：该层称为 Flink 应用框架层，根据 API 层的划分，在 API 层之上构建的满足特定应用的实现计算框架，也分别对应于面向流处理和面向批处理两类。面向流处理支持：CEP（复杂事件处理）、基于 SQL-like 的操作（基于 Table 的关系操作）；面向批处理支持：FlinkML（机器学习库）、Gelly（图处理）。

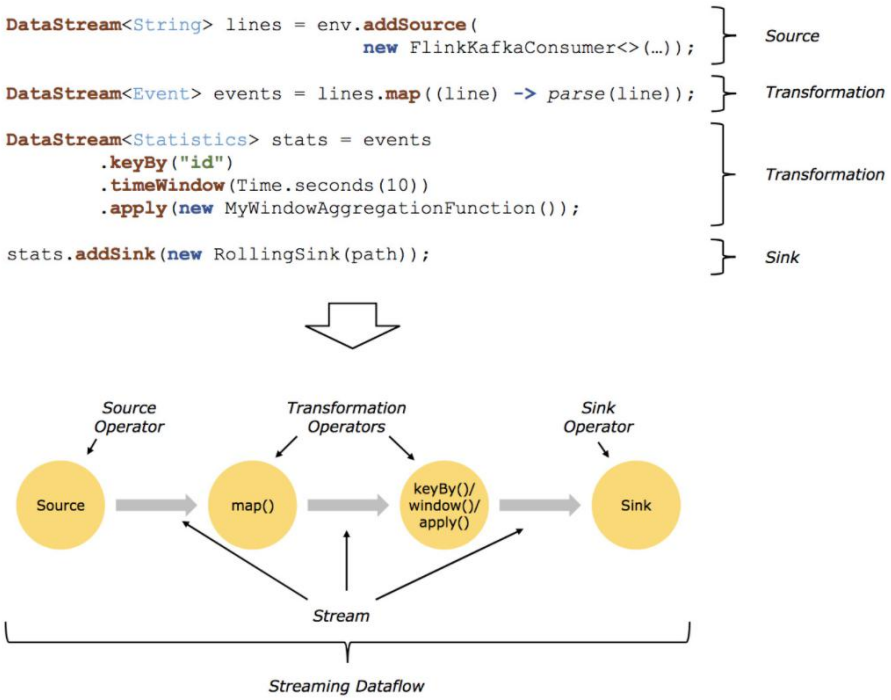
1.4 Flink 的运行必须依赖 Hadoop 组件吗？

Flink 可以完全独立于 Hadoop，在不依赖 Hadoop 组件下运行。但是做为大数据的基础设施，Hadoop 体系是任何大数据框架都绕不过去的。Flink 可以集成众多 Hadooop 组件，例如 Yarn、Hbase、HDFS 等等。例如，Flink 可以和 Yarn 集成做资源调度，也可以读写 HDFS，或者利用 HDFS 做检查点。

1.5 你们的 Flink 集群规模多大？

大家注意，这个问题看起来是问你实际应用中的 Flink 集群规模，其实还隐藏着另一个问题：Flink 可以支持多少节点的集群规模？在回答这个问题时候，可以将自己生产环节中的集群规模、节点、内存情况说明，同时说明部署模式（一般是 Flink on Yarn），除此之外，用户也可以同时在小集群（少于 5 个节点）和拥有 TB 级别状态的上千个节点上运行 Flink 任务。

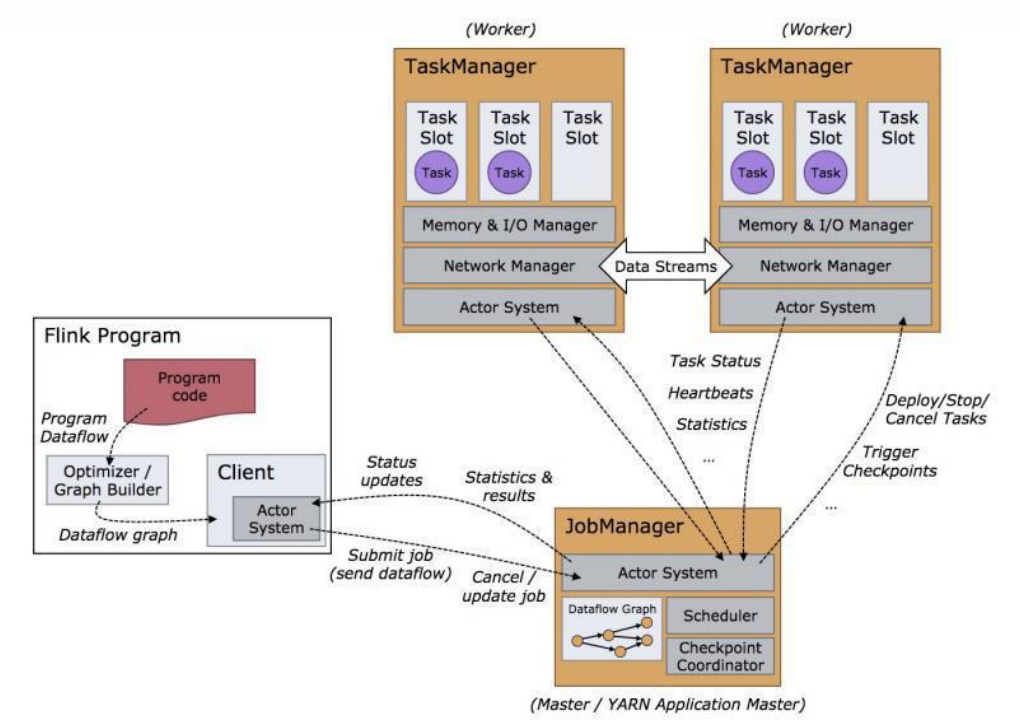
1.6 Flink 的基础编程模型了解吗？



上图是来自 Flink 官网的运行流程图。通过上图我们可以得知，Flink 程序的基本构建是数据输入来自一个 Source，Source 代表数据的输入端，经过 Transformation 进行转换，然后在一个或者多个 Sink 接收器中结束。数据流（stream）就是一组永远不会停止的数据记录流，而转换（transformation）是将一个或多个流作为输入，并生成一个或多个输出流的操作。执行时，Flink 程序映射到 streaming dataflows，由流（streams）和转换操作（transformation

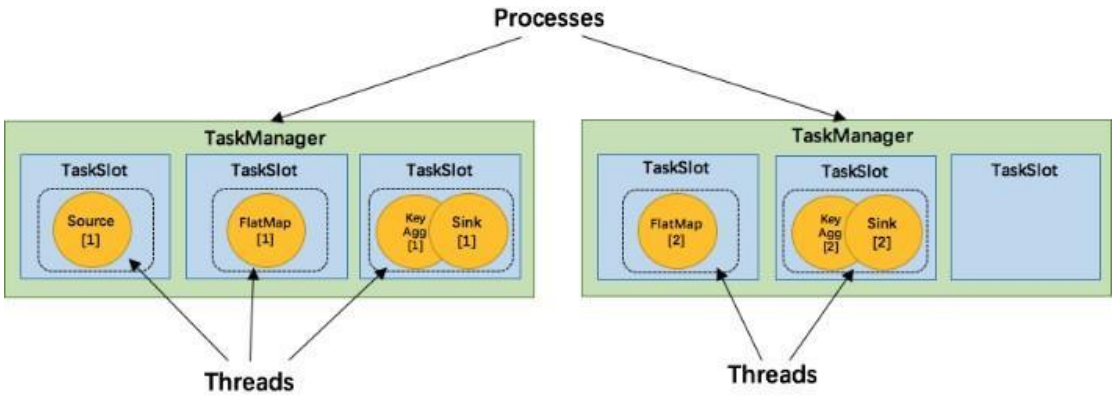
operators）组成。

1.7 Flink 集群有哪些角色？各自有什么作用？



Flink 程序在运行时主要有 TaskManager，JobManager，Client 三种角色。其中 JobManager 扮演着集群中的管理者 Master 的角色，它是整个集群的协调者，负责接收 Flink Job，协调检查点，Failover 故障恢复等，同时管理 Flink 集群中从节点 TaskManager。TaskManager 是实际负责执行计算的 Worker，在其上执行 Flink Job 的一组 Task，每个 TaskManager 负责管理其所在节点上的资源信息，如内存、磁盘、网络，在启动的时候将资源的状态向 JobManager 汇报。Client 是 Flink 程序提交的客户端，当用户提交一个 Flink 程序时，会首先创建一个 Client，该 Client 首先会对用户提交的 Flink 程序进行预处理，并提交到 Flink 集群中处理，所以 Client 需要从用户提交的 Flink 程序配置中获取 JobManager 的地址，并建立到 JobManager 的连接，将 Flink Job 提交给 JobManager。

1.8 说说 Flink 资源管理中 Task Slot 的概念



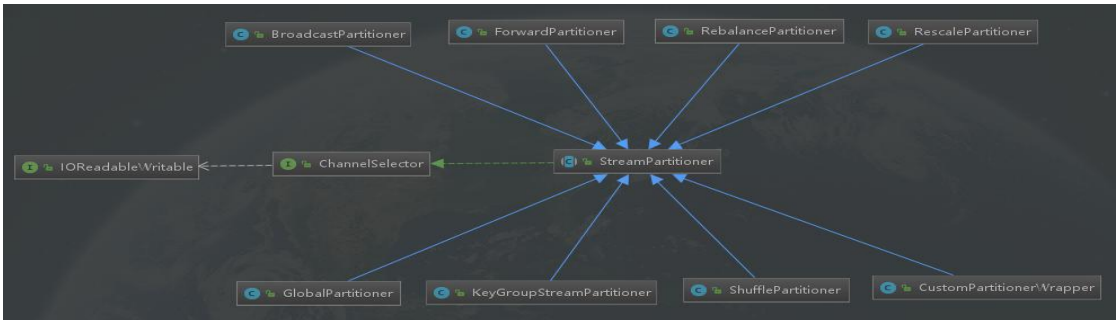
在 Flink 架构角色中我们提到，TaskManager 是实际负责执行计算的 Worker，TaskManager 是一个 JVM 进程，并会以独立的线程来执行一个 task 或多个 subtask。为了控制一个 TaskManager 能接受多少个 task，Flink 提出了 Task Slot 的概念。简单的说，TaskManager 会将自己节点上管理的资源分为不同的 Slot：固定大小的资源子集。这样就避免了不同 Job 的 Task 互相竞争内存资源，但是需要主要的是，Slot 只会做内存的隔离。没有做 CPU 的隔离。

1.9 说说 Flink 的常用算子？

Flink 最常用的常用算子包括：Map：DataStream → DataStream，输入一个参数产生一个参数，map 的功能是对输入的参数进行转换操作。Filter：过滤掉指定条件的数据。KeyBy：按照指定的 key 进行分组。Reduce：用来进行结果汇总合并。Window：窗口函数，根据某些特性将每个 key 的数据进行分组（例如：在 5s 内到达的数据）

1.10 说说你知道的 Flink 分区策略？

什么要搞懂什么是分区策略。分区策略是用来决定数据如何发送至下游。目前 Flink 支持了 8 中分区策略的实现。



上图是整个 Flink 实现的分区策略继承图：**GlobalPartitioner** 数据会被分发到下游算子的第

一个实例中进行处理。**ShufflePartitioner** 数据会被随机分发到下游算子的每一个实例中进行处理。**RebalancePartitioner** 数据会被循环发送到下游的每一个实例中进行处理。**RescalePartitioner** 这种分区器会根据上下游算子的并行度，循环的方式输出到下游算子的每个实例。这里有点难以理解，假设上游并行度为 2，编号为 A 和 B。下游并行度为 4，编号为 1，2，3，4。那么 A 则把数据循环发送给 1 和 2，B 则把数据循环发送给 3 和 4。假设上游并行度为 4，编号为 A，B，C，D。下游并行度为 2，编号为 1，2。那么 A 和 B 则把数据发送给 1，C 和 D 则把数据发送给 2。**BroadcastPartitioner** 广播分区会将上游数据输出到下游算子的每个实例中。适合于大数据集和小数据集做 Jion 的场景。**ForwardPartitioner** ForwardPartitioner 用于将记录输出到下游本地的算子实例。它要求上下游算子并行度一样。简单的说，ForwardPartitioner 用来做数据的控制台打印。**KeyGroupStreamPartitioner** Hash 分区器。会将数据按 Key 的 Hash 值输出到下游算子实例中。**CustomPartitionerWrapper** 用户自定义分区器。需要用户自己实现 Partitioner 接口，来定义自己的分区逻辑。例如：

```
static classCustomPartitionerimplementsPartitioner<String> {
      @Override
      publicintpartition(String key, int numPartitions) {
          switch (key){
              case "1":
                  return 1;
              case "2":
                  return 2;
              case "3":
                  return 3;
              default:
                  return 4;
          }
      }
  }
```

1.11 Flink 的并行度了解吗？Flink 的并行度设置是怎样的？

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。我们在实际生产环境中可以从四个不同层面设置并行度：

操作算子层面(Operator Level)

执行环境层面(Execution Environment Level)

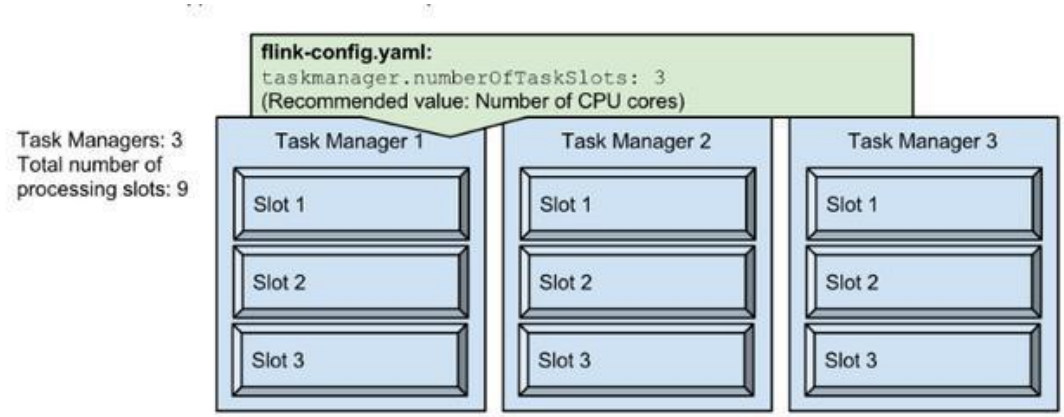
客户端层面(Client Level)

系统层面(System Level)

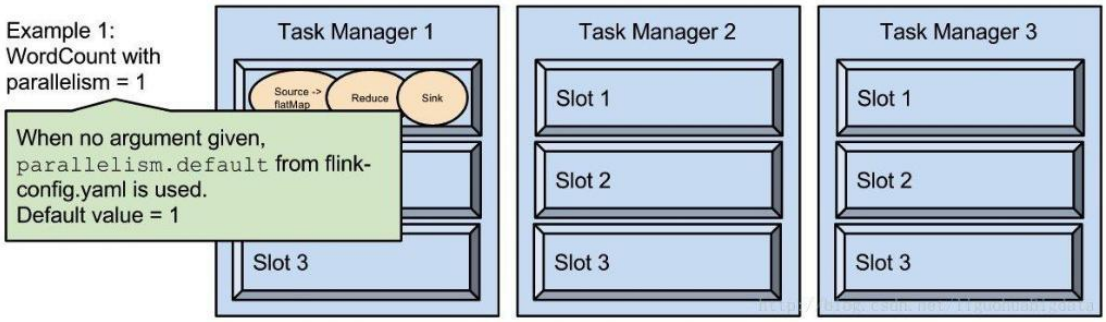
需要注意的优先级：算子层面>环境层面>客户端层面>系统层面。

1.12 Flink 的 Slot 和 parallelism 有什么区别？

官网上十分经典的图：



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。

1.13 Flink 有没有重启策略？说说有哪几种？

Flink 实现了多种重启策略。

固定延迟重启策略（Fixed Delay Restart Strategy）

故障率重启策略（Failure Rate Restart Strategy）

没有重启策略（No Restart Strategy）

Fallback 重启策略（Fallback Restart Strategy）

1.14 用过 Flink 中的分布式缓存吗？如何使用？

Flink 实现的分布式缓存和 Hadoop 有异曲同工之妙。目的是在本地读取文件，并把他放在 taskmanager 节点中，防止 task 重复拉取。

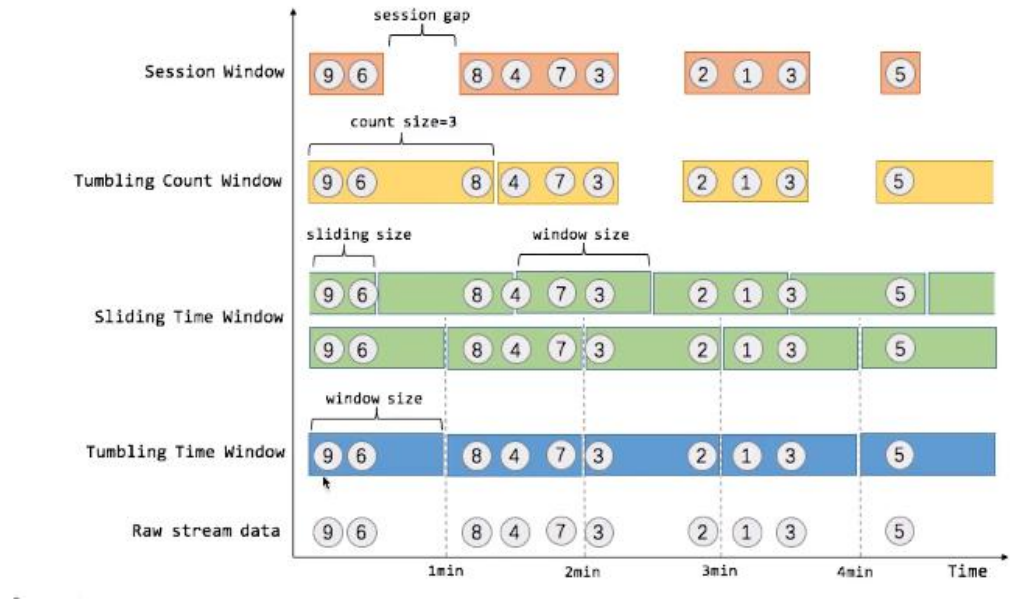
```
val env = ExecutionEnvironment.getExecutionEnvironment

// register a file from HDFS
env.registerCachedFile("hdfs:///path/to/your/file", "hdfsFile")

// register a local executable file (script, executable, ...)
env.registerCachedFile("file:///path/to/exec/file", "localExecFile",
true)

// define your program and execute
...
val input: DataSet[String] = ...
val result: DataSet[Integer] = input.map(new MyMapper())
...
env.execute()
```

2 说说 Flink 中的广播变量，使用时需要注意什么？

我们知道 Flink 是并行的，计算过程可能不在一个 Slot 中进行，那么有一种情况即：当我们需要访问同一份数据。那么 Flink 中的广播变量就是为了解决这种情况。我们可以把广播变量理解为是一个公共的共享变量，我们可以把一个 dataset 数据集广播出去，然后不同的 task 在节点上都能够获取到，这个数据在每个节点上只会存在一份。

3 说说 Flink 中的窗口？

来一张官网经典的图：



Flink 支持两种划分窗口的方式，按照 time 和 count。如果根据时间划分窗口，那么它就是一个 time-window 如果根据数据划分窗口，那么它就是一个 count-window。flink 支持窗口的两个重要属性（size 和 interval）如果 size=interval,那么就会形成 tumbling-window(无重叠数据) 如果 size>interval,那么就会形成 sliding-window(有重叠数据) 如果 size< interval, 那么这种窗口将会丢失数据。比如每 5 秒钟，统计过去 3 秒的通过路口汽车的数据，将会漏掉 2 秒钟的数据。通过组合可以得出四种基本窗口：

time-tumbling-window 无重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5))

time-sliding-window 有重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5), Time.seconds(3))

count-tumbling-window 无重叠数据的数量窗口，设置方式举例：countWindow(5)

count-sliding-window 有重叠数据的数量窗口，设置方式举例：countWindow(5,3)

1.17 说说 Flink 中的状态存储？

Flink 在做计算的过程中经常需要存储中间状态，来避免数据丢失和状态恢复。选择的状态存储策略不同，会影响状态持久化如何和 checkpoint 交互。Flink 提供了三种状态存储方式：MemoryStateBackend、FsStateBackend、RocksDBStateBackend。

1.18 Flink 中的时间有哪几类

Flink 中的时间和其他流式计算系统的时间一样分为三类：事件时间，摄入时间，处理

时间三种。如果以 EventTime 为基准来定义时间窗口将形成 EventTimeWindow,要求消息本身就应该携带 EventTime。如果以 IngesingtTime 为基准来定义时间窗口将形成 IngestingTimeWindow,以 source 的 systemTime 为准。如果以 ProcessingTime 基准来定义时间窗口将形成 ProcessingTimeWindow，以 operator 的 systemTime 为准。

1.19 Flink 中水印是什么概念，起到什么作用？

Watermark 是 Apache Flink 为了处理 EventTime 窗口计算提出的一种机制, 本质上是一种时间戳。一般来讲 Watermark 经常和 Window 一起被用来处理乱序事件。

1.20 Flink Table & SQL 熟悉吗？TableEnvironment 这个类有什么作用

TableEnvironment 是 Table API 和 SQL 集成的核心概念。这个类主要用来：

在内部 catalog 中注册表

注册外部 catalog

执行 SQL 查询

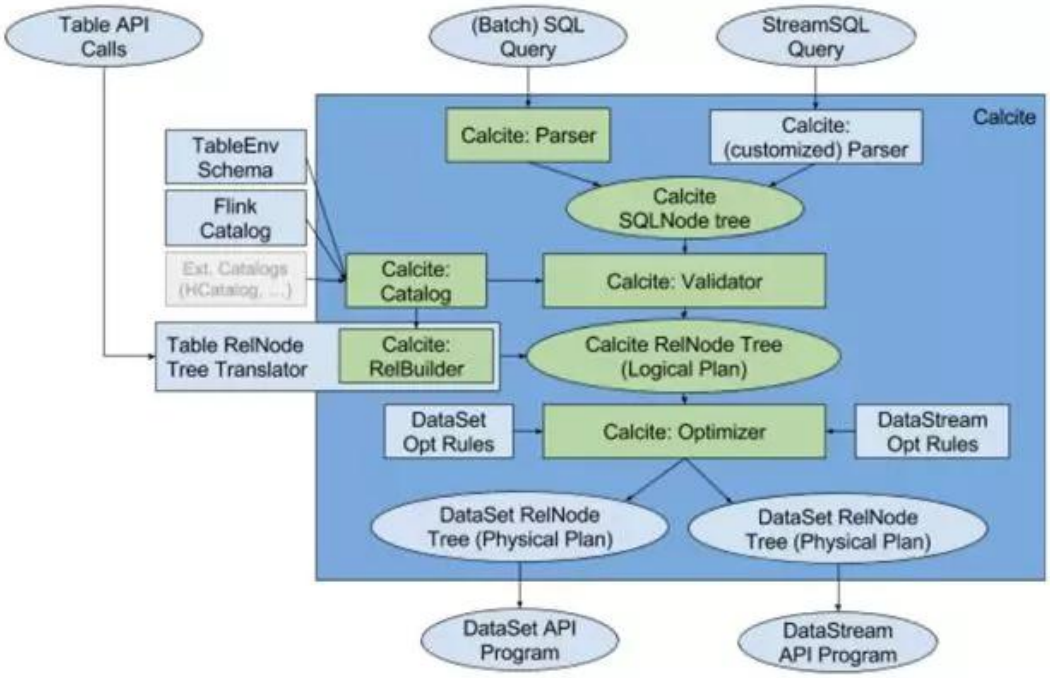
注册用户定义（标量，表或聚合）函数

将 DataStream 或 DataSet 转换为表

持有对 ExecutionEnvironment 或 StreamExecutionEnvironment 的引用

1.21 Flink SQL 的实现原理是什么？是如何实现 SQL 解析的呢？

首先大家要知道 Flink 的 SQL 解析是基于 Apache Calcite 这个开源框架。



基于此，一次完整的 SQL 解析过程如下：

用户使用对外提供 Stream SQL 的语法开发业务应用

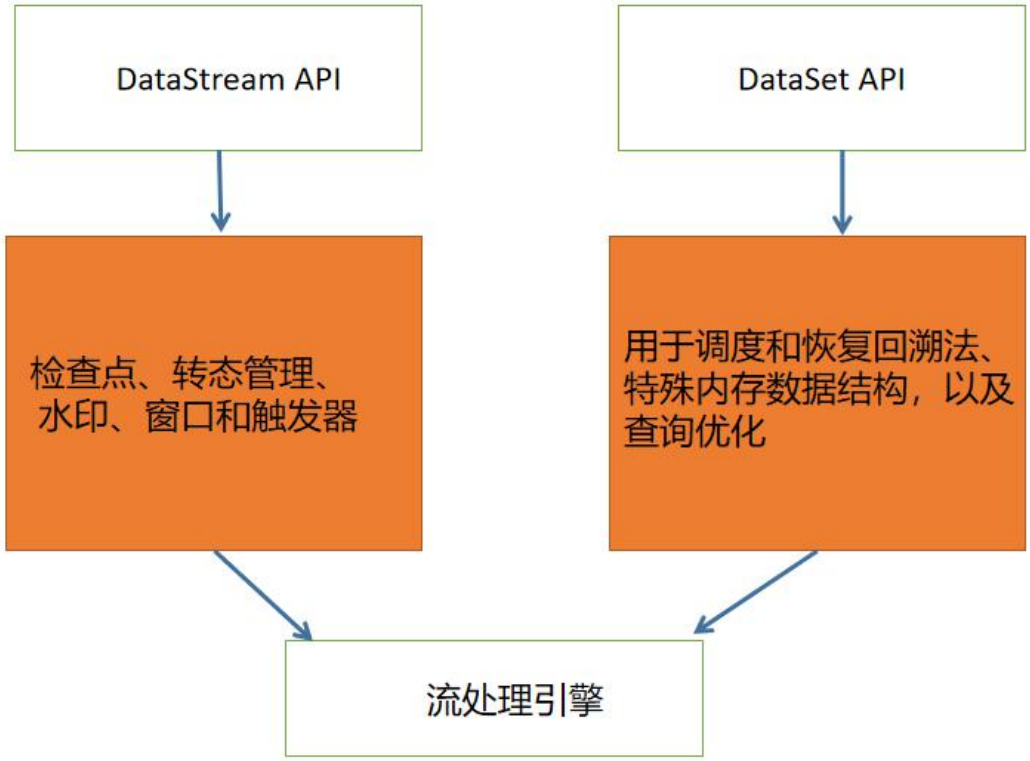
用 calcite 对 StreamSQL 进行语法检验，语法检验通过后，转换成 calcite 的逻辑树节点；最终形成 calcite 的逻辑计划

采用 Flink 自定义的优化规则和 calcite 火山模型、启发式模型共同对逻辑树进行优化，生成最优的 Flink 物理计划

对物理计划采用 janino codegen 生成代码，生成用低阶 API DataStream 描述的流应用，提交到 Flink 平台执行

2 Flink 中级

2.1 Flink 是如何支持批流一体的？



本道面试题考察的其实就是一句话：Flink 的开发者认为批处理是流处理的一种特殊情况。批处理是有限的流处理。Flink 使用一个引擎支持了 DataSet API 和 DataStream API。

2.2 Flink 是如何做到高效的数据交换的？

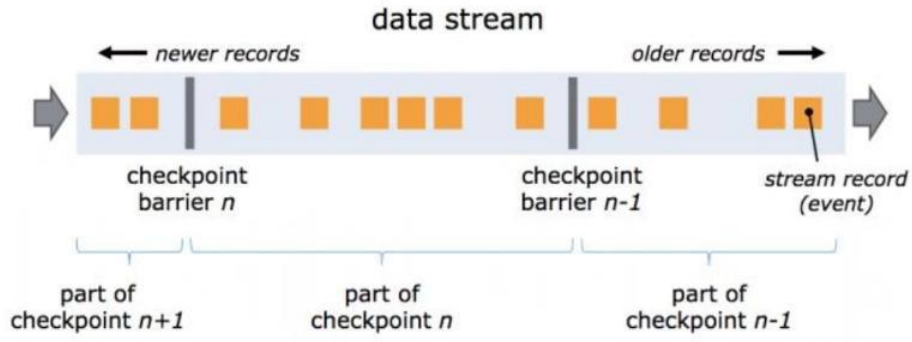
在一个 Flink Job 中，数据需要在不同的 task 中进行交换，整个数据交换是有 TaskManager 负责的，TaskManager 的网络组件首先从缓冲 buffer 中收集 records，然后再发送。Records 并不是一个一个被发送的，二是积累一个批次再发送，batch 技术可以更加高效的利用网络资源。

2.3 Flink 是如何做容错的？

Flink 实现容错主要靠强大的 CheckPoint 机制和 State 机制。Checkpoint 负责定时制作分布式快照、对程序中的状态进行备份；State 用来存储计算过程中的中间状态。

2.4 Flink 分布式快照的原理是什么？

Flink 的分布式快照是根据 Chandy-Lamport 算法量身定做的。简单来说就是持续创建分布式数据流及其状态的一致快照。



核心思想是在 input source 端插入 barrier，控制 barrier 的同步来实现 snapshot 的备份和 exactly-once 语义。

2.5 Flink 是如何保证 Exactly-once 语义的？

Flink 通过实现两阶段提交和状态保存来实现端到端的一致性语义。分为以下几个步骤：

开始事务（beginTransaction）创建一个临时文件夹，来写把数据写入到这个文件夹里面

预提交（preCommit）将内存中缓存的数据写入文件并关闭

正式提交（commit）将之前写完的临时文件放入目标目录下。这代表着最终的数据会有一些延迟

丢弃（abort）丢弃临时文件

若失败发生在预提交成功后，正式提交前。可以根据状态来提交预提交的数据，也可删除预提交的数据。

2.6 Flink 的 kafka 连接器有什么特别的地方？

Flink 源码中有一个独立的 connector 模块，所有的其他 connector 都依赖于此模块，Flink 在 1.9 版本发布的全新 kafka 连接器，摒弃了之前连接不同版本的 kafka 集群需要依赖不同版本的 connector 这种做法，只需要依赖一个 connector 即可。

2.7 说说 Flink 的内存管理是如何做的?

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。下面这段代码，根据配置文件中的各种参数来计算内存的分配方法。（heap or off-heap，这个放到下节谈），内存的分配支持预分配和 lazy load，默认懒加载的方式。

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。

2.8 说说 Flink 的序列化如何做的?

Java 本身自带的序列化和反序列化的功能，但是辅助信息占用空间比较大，在序列化对象时记录了过多的类信息。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(支持 Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的 Java Tuple 实现

CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)

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。

2.9 Flink 中的 Window 出现了数据倾斜，你有什么解决办法？

window 产生数据倾斜指的是数据在不同的窗口内堆积的数据量相差过多。本质上产生这种情况的原因是数据源头发送的数据量速度不同导致的。出现这种情况一般通过两种方式来解决：

在数据进入窗口前做预聚合

重新设计窗口聚合的 key

2.10 Flink 中在使用聚合函数 GroupBy、Distinct、KeyBy 等函数时出现数据热点该如何解决？

数据倾斜和数据热点是所有大数据框架绕不过去的问题。处理这类问题主要从 3 个方面入手：

在业务上规避这类问题

例如一个假设订单场景，北京和上海两个城市订单量增长几十倍，其余城市的数据量不变。这时候我们在进行聚合的时候，北京和上海就会出现数据堆积，我们可以单独数据北京和上海的数据。

Key 的设计上

把热 key 进行拆分，比如上个例子中的北京和上海，可以把北京和上海按照地区进行拆分聚合。

参数设置

Flink 1.9.0 SQL(Blink Planner) 性能优化中一项重要的改进就是升级了微批模型，即 MiniBatch。原理是缓存一定的数据后再触发处理，以减少对 State 的访问，从而提升吞吐和减少数据的输出量。

2.11 Flink 任务延迟高，想解决这个问题，你会如何入手？

在 Flink 的后台任务管理中，我们可以看到 Flink 的哪个算子和 task 出现了反压。最主要的手段是资源调优和算子调优。资源调优即是对作业中的 Operator 的并发数（parallelism）、CPU（core）、堆内存（heap_memory）等参数进行调优。作业参数调优包括：并行度的设置，State 的设置，checkpoint 的设置。

2.12 Flink 是如何处理反压的？

Flink 内部是基于 producer-consumer 模型来进行消息传递的，Flink 的反压设计也是基于这个模型。Flink 使用了高效有界的分布式阻塞队列，就像 Java 通用的阻塞队列（BlockingQueue）一样。下游消费者消费变慢，上游就会受到阻塞。

2.13 Flink 的反压和 Strom 有哪些不同？

Storm 是通过监控 Bolt 中的接收队列负载情况，如果超过高水位值就会将反压信息写到 Zookeeper ，Zookeeper 上的 watch 会通知该拓扑的所有 Worker 都进入反压状态，最后 Spout 停止发送 tuple。Flink 中的反压使用了高效有界的分布式阻塞队列，下游消费变

慢会导致发送端阻塞。二者最大的区别是 Flink 是逐级反压，而 Storm 是直接从源头降速。

2.14 Operator Chains（算子链）这个概念你了解吗？

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。这就是我们所说的算子链。

2.15 Flink 什么情况下才会把 Operator chain 在一起形成算子链？

两个 operator chain 在一起的的条件：

上下游的并行度一致

下游节点的入度为 1 （也就是说下游节点没有来自其他节点的输入）

上下游节点都在同一个 slot group 中（下面会解释 slot group）

下游节点的 chain 策略为 ALWAYS（可以与上下游链接，map、flatmap、filter 等默认是 ALWAYS）

上游节点的 chain 策略为 ALWAYS 或 HEAD（只能与下游链接，不能与上游链接，Source 默认是 HEAD）

两个节点间数据分区方式是 forward（参考理解数据流的分区）

用户没有禁用 chain

2.16 说说 Flink1.9 的新特性？

支持 hive 读写，支持 UDF

Flink SQL TopN 和 GroupBy 等优化

Checkpoint 跟 savepoint 针对实际业务场景做了优化

Flink state 查询

2.17 消费 kafka 数据的时候，如何处理脏数据？

可以在处理前加一个 fliter 算子，将不符合规则的数据过滤出去。

3 Flink 高级

3.1 Flink Job 的提交流程

用户提交的 Flink Job 会被转化成一个 DAG 任务运行，分别是：StreamGraph、JobGraph、ExecutionGraph，Flink 中 JobManager 与 TaskManager，JobManager 与 Client 的交互是基于

Akka 工具包的，是通过消息驱动。整个 Flink Job 的提交还包含着 ActorSystem 的创建，JobManager 的启动，TaskManager 的启动和注册。

3.2 Flink 所谓"三层图"结构是哪几个"图"？

一个 Flink 任务的 DAG 生成计算图大致经历以下三个过程：

StreamGraph 最接近代码所表达的逻辑层面的计算拓扑结构，按照用户代码的执行顺序向 StreamExecutionEnvironment 添加 StreamTransformation 构成流式图。

JobGraph 从 StreamGraph 生成，将可以串联合并的节点进行合并，设置节点之间的边，安排资源共享 slot 槽位和放置相关联的节点，上传任务所需的文件，设置检查点配置等。相当于经过部分初始化和优化处理的任务图。

ExecutionGraph 由 JobGraph 转换而来，包含了任务具体执行所需的内容，是最贴近底层实现的执行图。

3.3 JobManger 在集群中扮演了什么角色？

JobManager 负责整个 Flink 集群任务的调度以及资源的管理，从客户端中获取提交的应用，然后根据集群中 TaskManager 上 TaskSlot 的使用情况，为提交的应用分配相应的 TaskSlot 资源并命令 TaskManager 启动从客户端中获取的应用。JobManager 相当于整个集群的 Master 节点，且整个集群有且只有一个活跃的 JobManager ，负责整个集群的任务管理和资源管理。JobManager 和 TaskManager 之间通过 Actor System 进行通信，获取任务执行的情况并通过 Actor System 将应用的任务执行情况发送给客户端。同时在任务执行的过程中，Flink JobManager 会触发 Checkpoint 操作，每个 TaskManager 节点收到 Checkpoint 触发指令后，完成 Checkpoint 操作，所有的 Checkpoint 协调过程都是在 Fink JobManager 中完成。当任务完成后，Flink 会将任务执行的信息反馈给客户端，并且释放掉 TaskManager 中的资源以供下一次提交任务使用。

3.4 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。

3.5 TaskManager 在集群中扮演了什么角色？

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。

3.6 TaskManager 在集群启动过程中起到什么作用？

TaskManager 的启动流程较为简单：启动类：org.apache.flink.runtime.taskmanager.TaskManager 核心启动方法：selectNetworkInterfaceAndRunTaskManager 启动后直接向 JobManager 注册自己，注册完成后，进行部分模块的初始化。

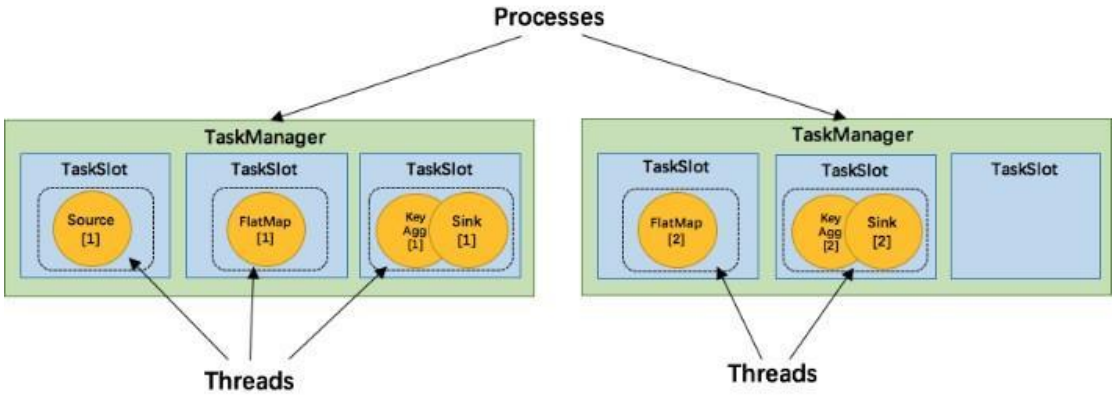
3.7 Flink 计算资源的调度是如何实现的？

TaskManager 中最细粒度的资源是 Task slot，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，如下图所示，FlatMap 函数占用一个 taskslot，而 key Agg 函数和 sink 函数共用一个 taskslot：

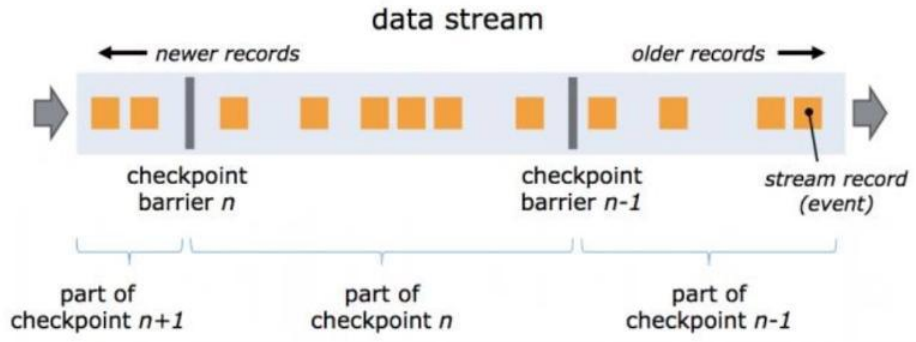


3.8 简述 Flink 的数据抽象及数据交换过程？

Flink 为了避免 JVM 的固有缺陷例如 java 对象存储密度低，FGC 影响吞吐和响应等，实现了自主管理内存。MemorySegment 就是 Flink 的内存抽象。默认情况下，一个 MemorySegment 可以被看做是一个 32kb 大的内存块的抽象。这块内存既可以是 JVM 里的一个 byte[]，也可以是堆外内存（DirectByteBuffer）。在 MemorySegment 这个抽象之上，Flink 在数据从 operator 内的数据对象在向 TaskManager 上转移，预备被发给下个节点的过程中，使用的抽象或者说内存对象是 Buffer。对接从 Java 对象转为 Buffer 的中间对象是另一个抽象 StreamRecord。

3.9 Flink 中的分布式快照机制是如何实现的？

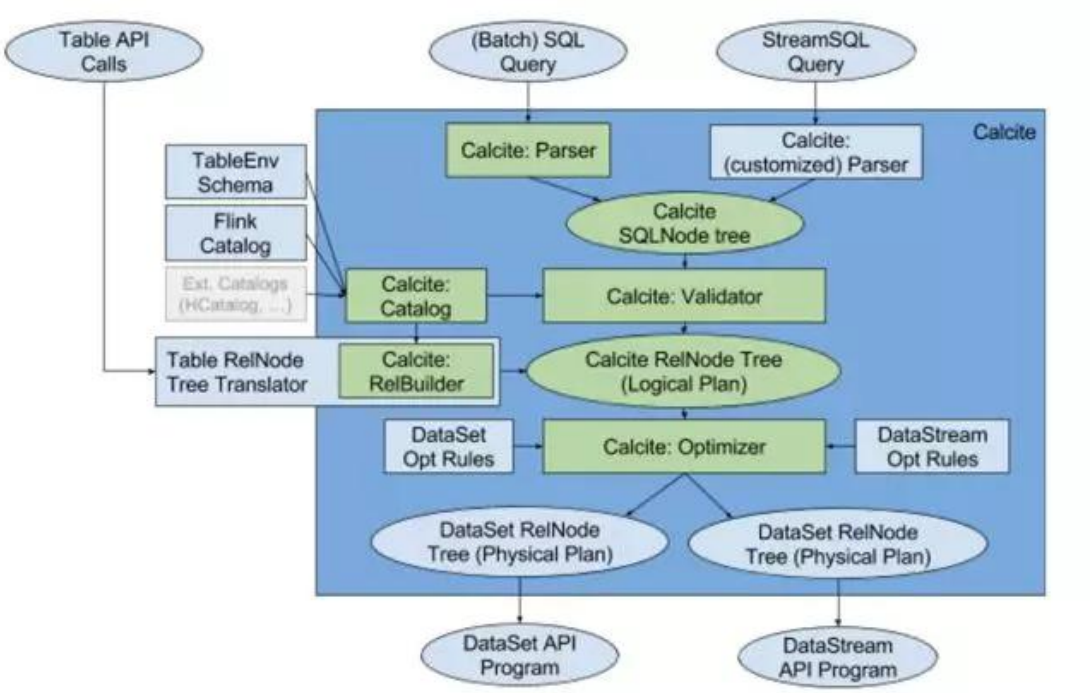
Flink 的容错机制的核心部分是制作分布式数据流和操作算子状态的一致性快照。 这些快照充当一致性 checkpoint，系统可以在发生故障时回滚。 Flink 用于制作这些快照的机制在“分布式数据流的轻量级异步快照”中进行了描述。 它受到分布式快照的标准 Chandy-Lamport 算法的启发，专门针对 Flink 的执行模型而定制。



barriers 在数据流源处被注入并行数据流中。快照 n 的 barriers 被插入的位置（我们称之为 Sn）是快照所包含的数据在数据源中最大位置。例如，在 Apache Kafka 中，此位置将是分区中最后一条记录的偏移量。将该位置 Sn 报告给 checkpoint 协调器（Flink 的 JobManager）。然后 barriers 向下游流动。当一个中间操作算子从其所有输入流中收到快照 n 的 barriers 时，它会为快照 n 发出 barriers 进入其所有输出流中。 一旦 sink 操作算子（流式 DAG 的末端）从其所有输入流接收到 barriers n，它就向 checkpoint 协调器确认快照 n 完成。在所有 sink 确认快照后，意味快照着已完成。一旦完成快照 n，job 将永远不再向数据源请求 Sn 之前的记录，因为此时这些记录（及其后续记录）将已经通过整个数据流拓扑，也即是已经被处理结束。

3.10 简单说说 FlinkSQL 的是如何实现的？

Flink 将 SQL 校验、SQL 解析以及 SQL 优化交给了 Apache Calcite。Calcite 在其他很多开源项目里也都应用到了，譬如 Apache Hive, Apache Drill, Apache Kylin, Cascading。Calcite 在新的架构中处于核心的地位，如下图所示。



构建抽象语法树的事情交给了 Calcite 去做。SQL query 会经过 Calcite 解析器转变成 SQL 节点树，通过验证后构建成 Calcite 的抽象语法树（也就是图中的 Logical Plan）。另一边，Table API 上的调用会构建成 Table API 的抽象语法树，并通过 Calcite 提供的 RelBuilder 转变成 Calcite 的抽象语法树。然后依次被转换成逻辑执行计划和物理执行计划。在提交任务后会分发到各个 TaskManager 中运行，在运行时会使用 Janino 编译器编译代码后运行。

4 企业面试

4.1 应用架构

问题：公司怎么提交的实时任务，有多少 Job Manager、Task Manager？

解答：

1. 我们使用 yarn session 模式提交任务；另一种方式是每次提交都会创 建一个新的 Flink 集群，为每一个 job 提供资源，任务之间互相独立，互不影响， 方便管理。任务执行完成之后创建的集群也会消失。线上命令脚本如下： bin/yarn-session.sh -n 7 -s 8 -jm 3072 -tm 32768 -qu root.*.* -nm *-* -d.其中申请 7 个 taskManager，每个 8 核，每个 taskmanager 有 32768M 内存。

2. 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。 对于 standlone 模式，我们公司一般配置一个主 Job Manager，两个备用 Job Manager，然后结合

ZooKeeper 的使用，来达到高可用；对于 yarn 模式，yarn 在 Job Mananger 故障会自动进行重启，所以只需要一个，我们配置的最大重启次数是 10 次。

4.2 压测和监控

问题：怎么做压力测试和监控？

解答：我们一般碰到的压力来自以下几个方面：

一，产生数据流的速度如果过快，而下游的算子消费不过来的话，会产生背压。 背压的监控可以使用 Flink Web UI(localhost:8081) 来可视化监控 Metrics，一旦报警 就能知道。一般情况下背压问题的产生可能是由于 sink 这个 操作符没有优化好， 做一下优化就可以了。比如如果是写入 ElasticSearch， 那么可以改成批量写入， 可以调大 ElasticSearch 队列的大小等等策略。

二，设置 watermark 的最大延迟时间这个参数，如果设置的过大，可能会造成 内存的压力。可以设置最大延迟时间小一些，然后把迟到元素发送到侧输出流中去。 晚一点更新结果。或者使用类似于 RocksDB 这样的状态后端， RocksDB 会开辟 堆外存储空间，但 IO 速度会变慢，需要权衡。

三，还有就是滑动窗口的长度如果过长，而滑动距离很短的话，Flink 的性能 会下降的很厉害。我们主要通过时间分片的方法，将每个元素只存入一个“重叠窗 口”，这样就可以减少窗口处理中状态的写入。

四，状态后端使用 RocksDB，还没有碰到被撑爆的问题。

4.3 有了 Spark 还为什么用 Flink

问题：为什么使用 Flink 替代 Spark？

解答：主要考虑的是 flink 的低延迟、高吞吐量和对流式数据应用场景更好的支 持；另外，flink 可以很好地处理乱序数据，而且可以保证 exactly-once 的状态一致 性。详见文档第一章，有 Flink 和 Spark 的详细对比。

4.4 checkpoint 的存储

问题：Flink 的 checkpoint 存在哪里？

解答：可以是内存，文件系统，或者 RocksDB。

- 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

4.5 exactly-once 的保证

问题：如果下级存储不支持事务，Flink 怎么保证 exactly-once？

解答：端到端的 exactly-once 对 sink 要求比较高，具体实现主要有幂等写入和 事务性写入两种方式。幂等写入的场景依赖于业务逻辑，更常见的是用事务性写入。 而事务性写入又有预写日志（WAL）和两阶段提交（2PC）两种方式。 如果外部系统不支持事务，那么可以用预写日志的方式，把结果数据先当成状 态保存，然后在收到 checkpoint 完成的通知时，一次性写入 sink 系统。

4.6 状态机制

问题：说一下 Flink 状态机制？

解答：Flink 内置的很多算子，包括源 source，数据存储 sink 都是有状态的。在 Flink 中，状态始终与特定算子相关联。Flink 会以 checkpoint 的形式对各个任务的 状态进行快照，用于保证故障恢复时的状态一致性。Flink 通过状态后端来管理状态 和 checkpoint 的存储，状态后端可以有不同的配置选择。

4.7 海量 key 去重

问题：怎么去重？考虑一个实时场景：双十一场景，滑动窗口长度为 1 小时， 滑动距离为 10 秒钟，亿级用户，怎样计算 UV？

解答：使用类似于 scala 的 set 数据结构或者 redis 的 set 显然是不行的， 因为可能有上亿个 Key，内存放不下。所以可以考虑使用布隆过滤器（Bloom Filter） 来去重。

4.8 checkpoint 与 spark 比较

问题：Flink 的 checkpoint 机制对比 spark 有什么不同和优势

解答：spark streaming 的 checkpoint 仅仅是针对 driver 的故障恢复做了数据 和元数据的 checkpoint。而 flink 的 checkpoint 机制 要复杂了很多，它采用的是 轻量级的分布式快照，实现了每个算子的快照，及流动中的数据的快照.

4.9 watermark 机制

问题：请详细解释一下 Flink 的 Watermark 机制。

解答：Watermark 本质是 Flink 中衡量 EventTime 进展的一个机制，主要用来处 理乱序数据。

4.10 exactly-once 如何实现

问题：Flink 中 exactly-once 语义是如何实现的，状态是如何存储的？

解答：Flink 依靠 checkpoint 机制来实现 exactly-once 语义，如果要实现端到端 的 exactly-once，还需要外部 source 和 sink 满足一定的条件。状态的存储通过状态 后端来管理，Flink 中可以配置不同的状态后端。

4.11 CEP

问题：Flink CEP 编程中当状态没有到达的时候会将数据保存在哪里？

解答：在流式处理中，CEP 当然是要支持 EventTime 的，那么相对应的也要 支持数据的迟到现象，也就是 watermark 的处理逻辑。CEP 对未匹配成功的事件序 列的处理，和迟到数据是类似的。在 Flink CEP 的处理逻辑中，状态没有满足的和 迟到的数据，都会存储在一个 Map 数据结构中，也就是说，如果我们限定判断事件 序列的时长为 5 分钟，那么内存中就会存储 5 分钟的数据，这在我看来，也是对内 存的极大损伤之一。

4.12 三种时间语义

问题：Flink 三种时间语义是什么，分别说出应用场景？

解答：

1. Event Time ： 这是实际应用最常见的时间语义

2. Processing Time：没有事件时间的情况下，或者对实时性要求超高的情况下。

3. Ingestion Time：存在多个 Source Operator 的情况下，每个 Source Operator 可以使用自己本地系统时钟指派 Ingestion Time。后续基于时间相关的各种操作， 都会使用数据记录中的 Ingestion Time。

4.13 数据高峰的处理

问题：Flink 程序在面对数据高峰期时如何处理？

解答：使用大容量的 Kafka 把数据先放到消息队列里面作为数据源，再使用 Flink 进行消费，不过这样会影响到一点实时性。