Apache Flink

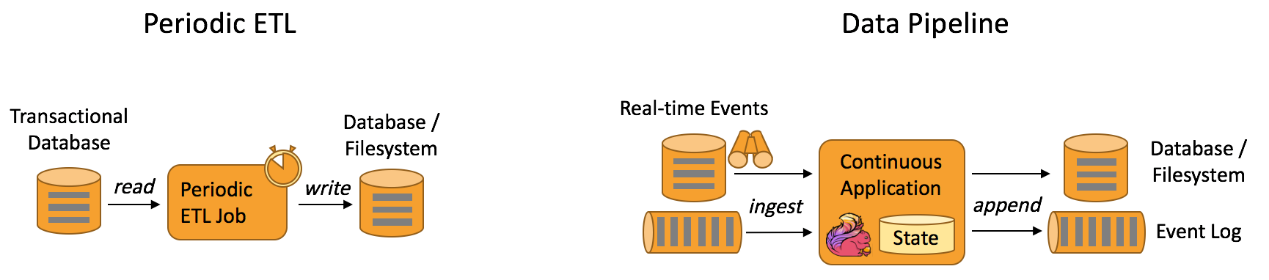
Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale.

## Flink能做什么？

* 实际的生产过程中，对大量的数据进行实时分析 -- > 实时计算、实时推荐、实时预警(黄牛刷单、盗号支付)等等。
* 传统方式：各业务服务数据落库，如订单、库存、物流、支付等等，然后基于已落库的数据拉宽表，汇总后做离线分析，时效性太差(如刷单封号、支付异常冻结等)。
* 大量（海量数据）：N个业务系统产生大量的数据 -> 支持大数据量处理
* 实时（时效性）：业务实时可视 -> 快速处理，实时计算；如，数据采集->校验->转换->关联->计算(归并&拆分&汇总&聚合&分组…)->分析->结果，秒级可视

## Flink应用场景

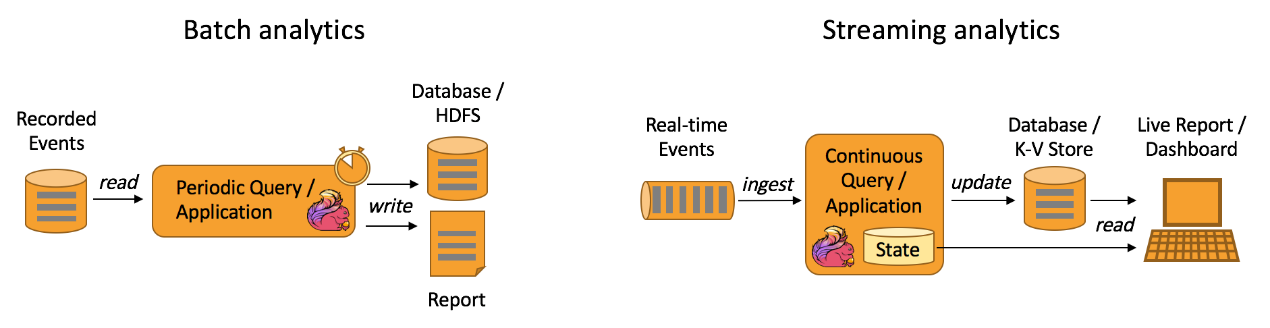
* Data Pipeline Applications：数据搬运并在搬运过程中进行数据清洗、处理；如上游对接Kafka，消息输入后进行数据清洗，然后将处理结果输出至下游的Database或者File system。举例：
  + 实时数仓：实时采集->实时处理->实时查询，保证数据查询的时效性
  + 搜索引擎推荐：店铺上新品->消息->kafka->flink->数据整理(如数据扩展、索引数据组装等)->搜索引擎创建索引­->秒级可试



传统批处理(左)：ETL作业通常会周期性地触发，将数据从事务型数据库拷贝到分析型数据库或数据仓库。

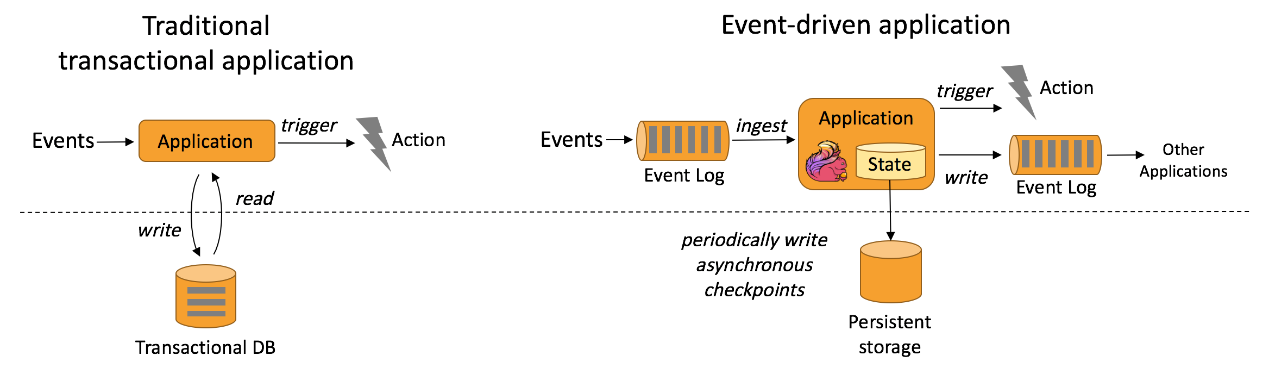
实时流处理(右)：以持续流模式运行，而非周期性触发。因此它支持从一个不断生成数据的源头读取记录，并将它们以低延迟移动到终点。

* Data Analytics [ˌænəˈlɪtɪks] Applications：流批一体 -> 实时大屏、实时报表



左边: 离线分析，批处理 –> 报表、大屏；右边: 实时分析，流处理 –> 报表、大屏

* Data-driven Applications：将各种规则内置于flink之中（如Datastream API），当某个事件进入后，会触发规则的执行，如通知相关业务系统，做出预警，适合做风控系统



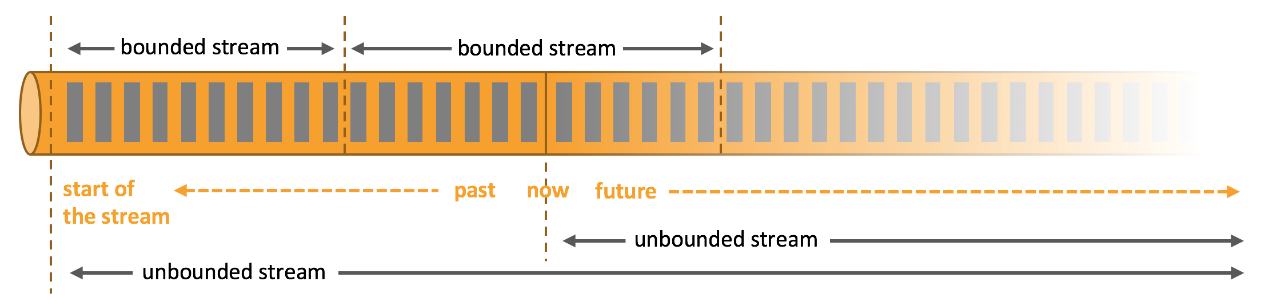
而传统的做法是：事件触发->读事务型数据库->计算分析->回写事务型数据库->发通知

## 框架对比

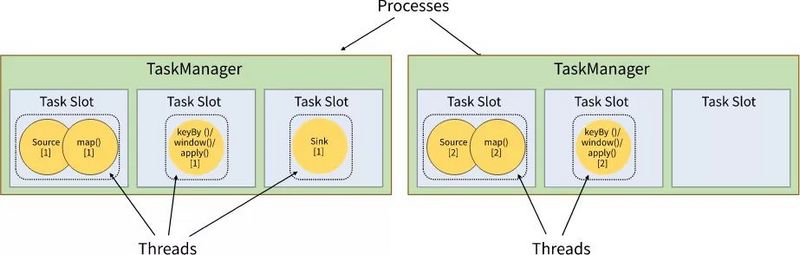
* 批处理：Hadoop，以MapReduce作为原生处理引擎，以HDFS作为分布式文件存储，以YARN(Yet Another Resource Negotiator，另一个资源管理器)作为集群协调
* 流处理： Storm，侧重极低延迟的流处理框架，时效性最好的流处理框架, At least once
* 流批一体：一套逻辑，通过缓存块超时值来实现流、批一体的处理逻辑
  + Spark：包含流处理的下一代批处理框架，可单独替换Hadoop集群中的MapReduce引擎作为其批处理引擎；其核心引擎是面向“批处理”概念的,不是一款纯流式计算引擎,在时效性等问题上无法提供极致的流批一体体验；但Spark基于一套核心引擎技术,同时实现流和批两种计算语义的理念是非常先进的。Spark的流处理其本质还是批处理，其内部采用了一种micro-batch的架构，即把输入的数据流切分成了细粒度的batch，然后为每一个batch提交一个批处理的任务(比如每秒处理一批)，来达到流处理的效果，其和Storm等完全流处理的方式完全不同。
  + Flink：侧重低延迟流处理(性能同Storm差不多)，支持少量批处理；支持增量计算；Flink 的理念和目标也是利用一套计算引擎同时支持流和批两种计算模式,但它和 Spark 相比选择了不同的实现路线。Flink 选择了面向“流处理”的引擎架构,并认为“批”其实是一种“有限流”,基于流为核心的引擎实现流批一体更加自然,并且不会有架构瓶颈,可以认为Flink选择了batch on streaming的架构,不同于Spark选择的streaming on batch架构。流批的实现原理：引入缓存块
    - 缓存块超时值=0，A节点处理数据->缓存->B节点立即处理
    - 缓存块超时值=无穷大，A节点处理N条数据->缓存->A继续接收数据处理->缓存->…->没数据了->B节点再处理
    - 缓存块超时值=0~无穷大，A节点处理N条数据->缓存->达到超时值->B节点处理
* 流计算在google dataflow中被抽象为四个问题：what、where、when、how？
* 存储
  + HDFS: Hadoop Distributed File System，Hadoop分布式文件存储系统，提供了一个具有高度容错性和高吞吐量的海量数据存储解决方案。存储大文件优势明显，小文件的话可能反而性能不高。
  + HBase：Hadoop database，也即是Hadoop数据库，是一种NoSql面向列存储的数据库。主要适用于海量明细数据的随机实时查询，如日志、交易明细清单，行为轨迹等等。不支持SQL操作。
  + Hive：Hadoop的数据仓库，严格来说不是数据库。主要是让开发者能够通过SQL来计算和处理HDFS上的结构化数据，适用于离线的批量数据计算。如，架于HBase之上，可以以SQL方式操作HBase.
  + ClickHouse: 一款高性能列式存储数据库，属于OLAP型数据库；某些场景，也可拿来代替ES，如日志的检索；

## 概念

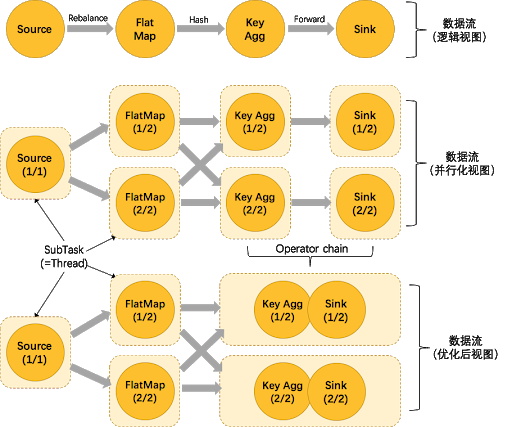
* **Streams（Dataflow）**：
  + 无界数据流-源源不断的消息，有始无终，计算持续进行且不存在结束状态；有界数据流-数据库某时刻的数据快照，有始有终，计算最终会完成并处于结束状态，有界流的处理也称为批处理。



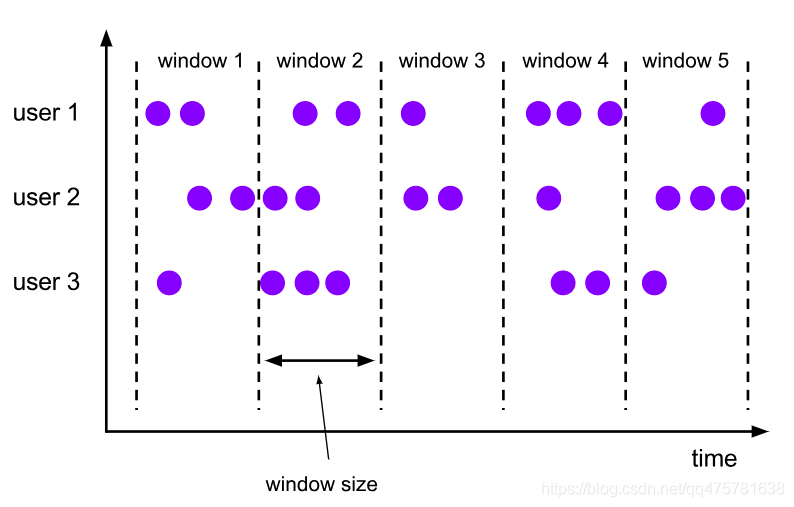
* + Dataflow也是一个分布式概念，它的stream被拆分成了Stream-partition(其本质就是Data-partition)
* **JobManager**：
  + 独立进程；
  + 大管家，flink集群中至少启动一个实例，调度管控中心；- 管理checkpoint、容灾、故障恢复等
  + 主从部署（master-slave based on zookeeper， JM->master）,至少保证一个active master，其它standby
* **TaskManager**：
  + 独立进程；
  + 任务执行者，接收来自JobManager的task
  + 受JobManager管控；自动将心跳和统计信息汇报给JobManager；
  + 主从部署（master-slave based on zookeeper， TM->slave），N个TM实例组成worker集群
* **TaskSlot：**

****

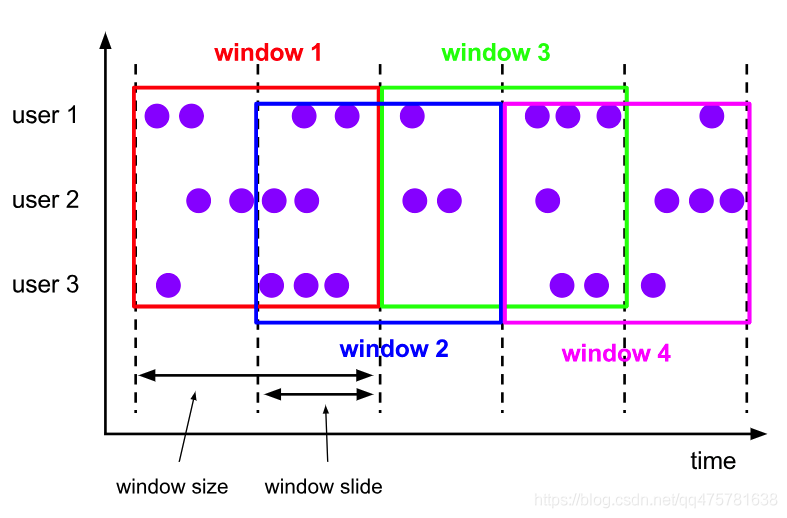
* + JobManager在启动的时候，就设置好了槽(Slot)位数，Slot之间会隔离内存资源，不会隔离CPU资源；如一个TaskManager分了4个Slot，那么它将为每个Slot分配25%的内存；
  + 类比java lambda理解
* 需求：【获取用户列表(datasource)】->【映射(user->name)】->【去重】->【排序】->【连接】 组成一个job，然后交给TaskManager执行；代码流程如下：users.parallelStream().map(User::getName).distinct().sorted(Comparator.naturalOrder()).collect(Collectors.joining(",", "[", "]"))，
* 如上Job包含4个task,5个operator，可(能)优化为(flink生成执行计划):
* Task1-users.parallelStream()；task:operator(subtask) = 1:1
* Task2-map(User::getName)；task:operator(subtask) = 1:1
* Task3-distinct().sorted(Comparator.naturalOrder())；task:operator(subtask) = 1:2
* Task4-.collect(Collectors.joining(",", "[", "]"))；task:operator(subtask) = 1:1
* map()、distinct ()、sorted ()等等即为一个个的operator，每个operator都可以设置parallelism来并行执行，如map(User::getName)可以被N个线程同时执行，加快map转换速度，最后的效果是map()被N个线程执行(可夸机器,flink天然分布式计算)
  + Task是执行的最小单位，一个Task可以处理多个Operator(即一个task包含N个subtask)，一般这些Operator是能被Chain在一起处理的，称为OperatorChain,也可称为task pipeline。如下图统计单词出现的数量，最后两步keyBy()&sink()操作完全一个线程处理即可，没必要再单独起一个线程做输出操作，耗费CPU



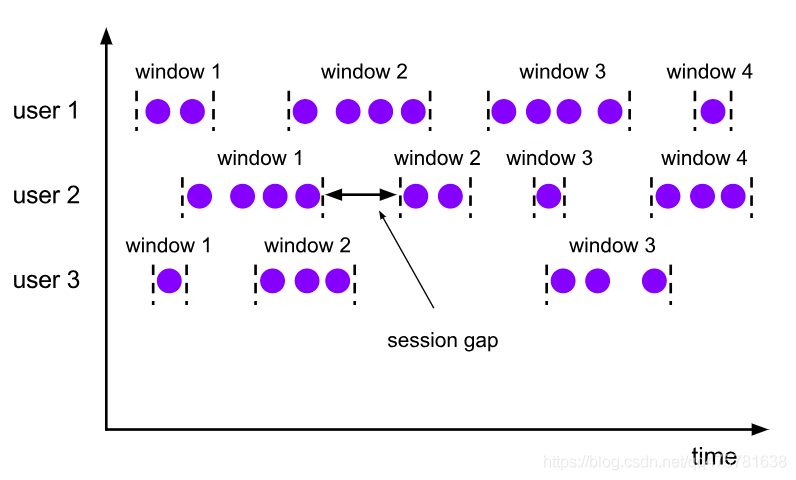
* + SlotSharingGroup：不同的task可以共享slot。
  + CoLocationGroup：
* **Parallelism:** 并行度，直白点就是多线程。一个Flink程序是由多个task组成（source、transformation和sink）。一个task由多个并行的实例（多线程）来执行，一个task并行实例（线程）数目就被称为该任务的并行度。并行的意思，在Flink中代表每个task的并行度，适当的提高并行度可以大大提高job的执行效率，比如当你的job消费kafka的速度过慢，适当调大就消费正常了，即多线程消费。Flink默认并行度为1，即单线程顺序消费，如一个线程每次消费一条消息，每条消息顺序执行input->filter->map->distinct…->output。任务的并行度可从多个从层指定，operator级别设置 > task级别设置 > Execution Environment 级别设置> flink默认设置
* **Client**：建立与JobManager的连接，任务提交至JobManager，再由JobManager分配任务至各TaskManager去执行；独立进程；client将任务提交后，可以选择结束进程，亦可选择等待结果返回；clinet属于flink架构的一部分，但不属于flink集群
* **Akka：**数据通信方式， Akka是一个用scala构建的支持高并发、分布式和容错的工具包，其处理并发的模式基于Actor，Actor之间通信的唯一机制就是消息投递，Actor解决了java中锁、原子性等弊端线程安全问题；TaskManager之间以流的形式进行数据的传输(基于Netty)；
* **Time:** Flink的无界流是一个持续的过程，时间是我们判断业务状态是否滞后、数据处理是否及时的重要依据；一般情况下我们优先关注Event Time，在Event Time无法使用时，在考虑其他类型的Time
  + Event time: the time when an event created. 如用户点击某个页面或者按钮的时间,在进入flink之前就已经被确定了
  + Ingestion time: the time when an event enters the flink dataflow at the source operator. 进入flink的时间
  + Processing time: 事件被处理时当前机器的时间
* **Window:** 处理无界流的核心，将流拆分为有限大小的“桶(buckets)”，然后在其上做计算；window是流到批的一个桥梁；窗口可以是基于时间驱动的，如每隔30s；也可以是基于事件数量驱动的，如每100个元素；也可以是基于会话间隔驱动的，其分为三类：
  + Tumbling Windows：滚动(翻滚、固定、无重合)窗口。固定大小，元素不会重复，一条记录只属于一个窗口。可简单理解为一个正方体在一直往前滚动。



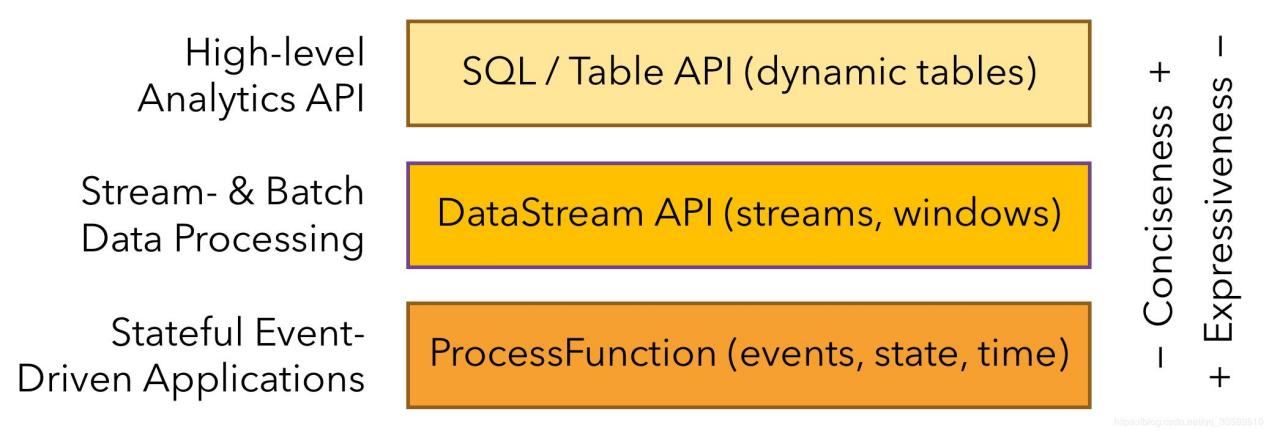
* + Sliding Windows：滑动窗口。固定大小，元素会重复，一条记录属于多个窗口。可理解为每隔一分钟统计过去两分钟的数据，即1&2 -> 2&3 -> 3&4...



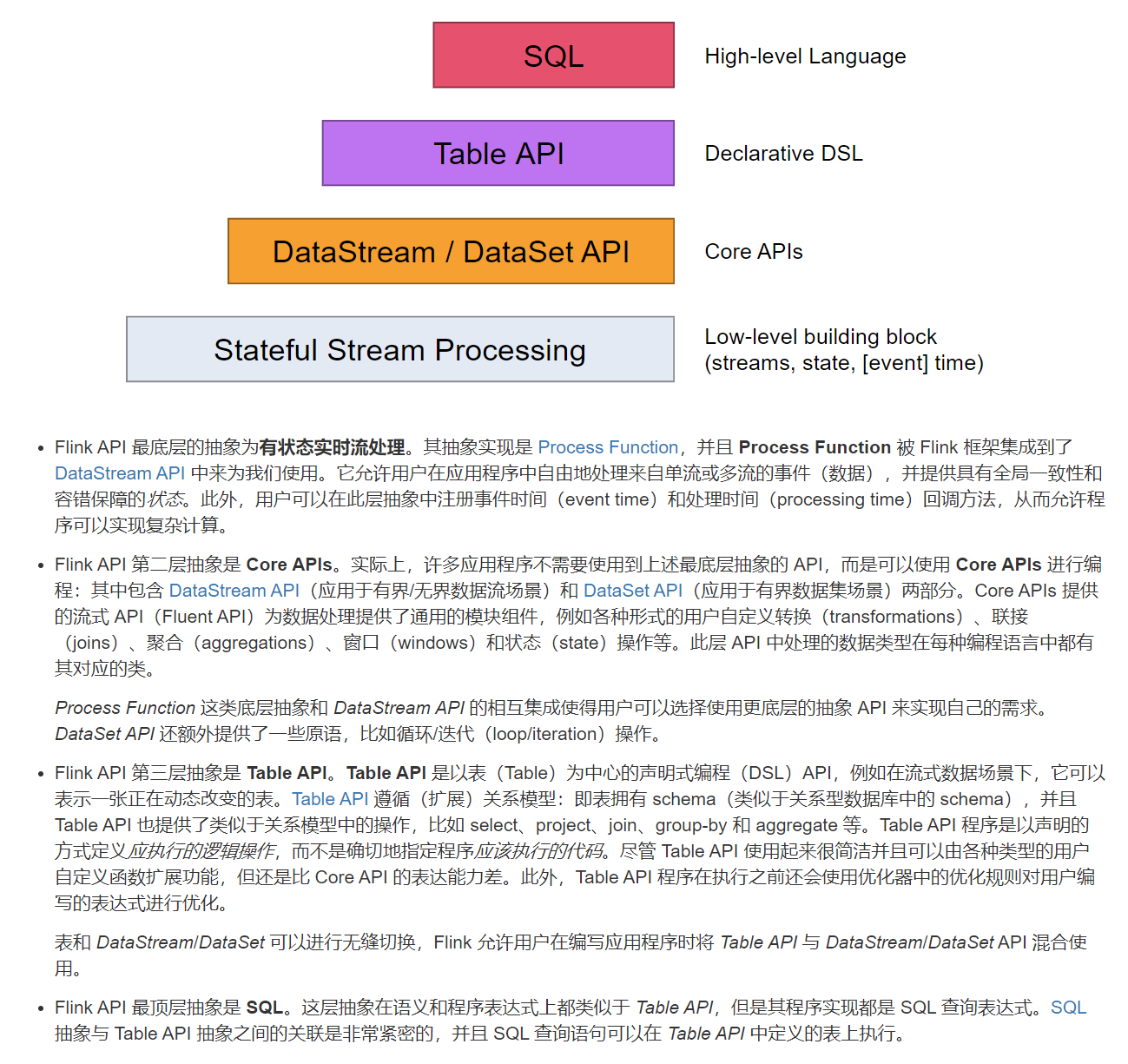
* + Session Windows：会话窗口。类比于tomcat session，创建一个session窗口不指定开始结束时间，一直接收数据，如果N段时间内(session gap)无数据流入，那么结束前一个session窗口做计算，然后新建一个session窗口接收新的数据。元素不会重复，没有固定的开始/结束时间，一条记录一个窗口。



* **Watermark & Trigger:** 
  + 问题：由于网络传输、服务可用性等问题，导致先产生的event晚于后产生的event到达Flink processor
  + Watermark用来标记窗口的完整性;Trigger用来设计窗口数据触发条件。
  + Watermark是基于已收集的消息来估算是否还有消息未到达，本质上是一个时间戳。时间戳反映的是事件发生的时间，而不是事件处理的时间。
  + 当数据进入flink系统的时候，我们基于事件的Event time创建window以后，我们如何确认这个window的数据是否已经全部到达，如果已经全部到达那么就可以进行窗口计算操作，如果没有全部到达，那么就需要进行等待数据的全部到达。但是我们对于迟到的数据不能够无休止的等下去，所有我们必须有一个机制来保证特定的时间后，就必须出发window的计算操作，这个就是Watermark机制,其作用就是告知算子之后不会有小于或等于水印时间戳的事件。
* **API: SQL/Table API -> DataStream API -> ProcessFunction三层，**越接近SQL/Table API表达能力越弱 ，抽象能力越强(关系型API)；相反，越接近ProcessFunction,表达能力越强，抽象能力越弱。类比于web应用的api、service、dao三层结构。SQL/Table API->用户只需要关心做什么，不需要关心怎么做，语义明确，易理解，所见即所得。Table API 是 Flink 自身的一套 API，这使得我们更容易地去扩展标准的 SQL，对比 SQL，我们可以认为 Table API 是 SQL 的超集。SQL 有的操作，Table API 可以有，然而我们又可以从易用性和功能性地角度对 SQL 进行扩展和提升。
  + **三层**



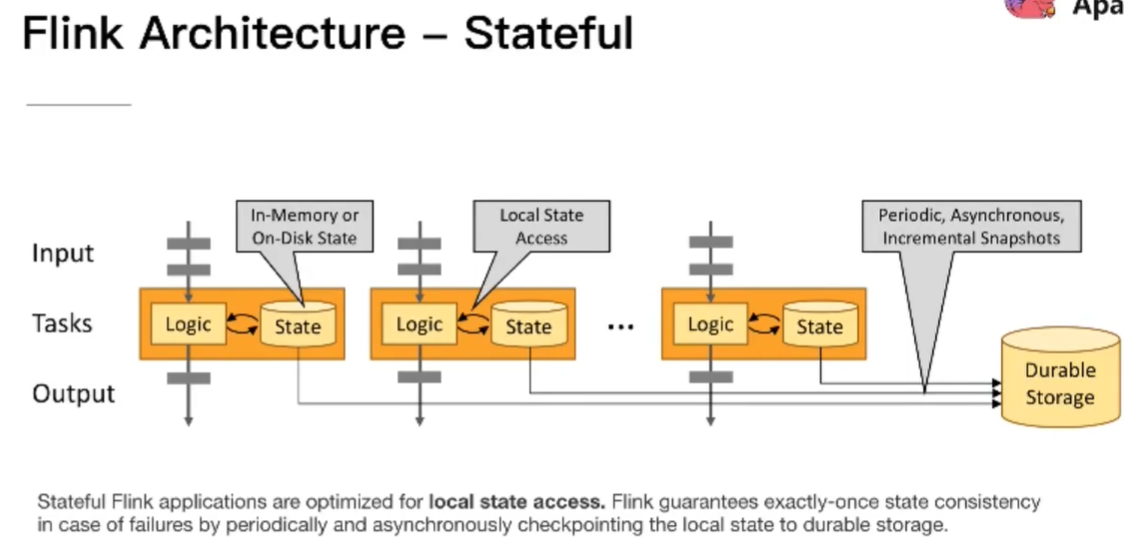
* + **四层：越往下越灵活，越往上越**



* **Exactly-once:** 在任何情况下，都能保证数据对应用产生的效果只有一次，不会多也不会少。
* **State:** 流计算本质上来讲是Incremental processing,因此要不断查询保持状态来保证其逻辑处理。stateless(无状态):如web应用的http请求，处理完不保存数据直接返回，下一个http请求与上一个没任何关系；stateful(有状态): 意味着数据要不断的流入并保存，如需要计算过去一段时间的PV，那么则需要将过去一段时间的数据保存下来然后才能计算，称之为有状态。异步备份，如果某个节点挂掉了，其它节点可以接着前序状态继续计算，保证Exactly-once semantics。

state一般指一个具体的task或者operator的状态。

* Operator State：普通的操作状态，如map()、reduce()等等
* Keyed State：分组后的基于KeyedStream的状态，这个状态跟key是绑定的；即KeyedStream上的每一个key都能对应一个state
* 原始状态&托管状态：Operator State和Keyed Stat可以以两种形式存在，即原始状态和托管状态；原始­->序列化&反序列化；托管->保存状态至hdfs&从hdfs恢复出来



* **Checkpoint:** 
  + 保证作业失败的情况下从最近一次快照进行恢复，从而保证系统内部的Exactly-once；Flink实现容错机制的核心；直白一点理解就是在某一时刻，将所有task的状态做一个snapshot存储，便于后续自动回复，无需人工干预
  + 原理：？？？重要！！！
* **Savepoint：**一种特殊的checkpoint，用于手动恢复，一般不需要

## 三大步骤

Data Source(数据源头) 🡪

Transformation(算子计算, map/flatmap/filter/groupBy/keyBy/reduce/sum…) 🡪

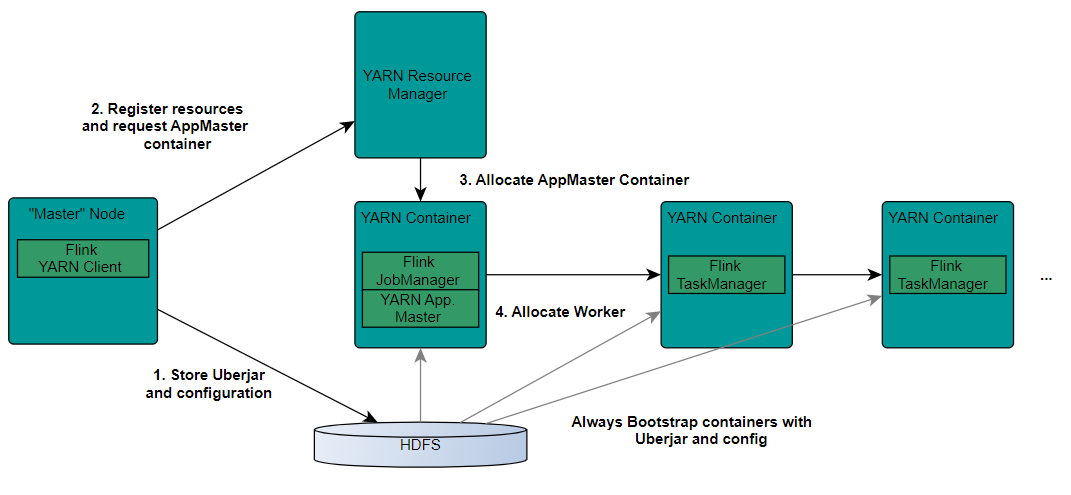
Data Sink(输出结果, Kafka/ES/Mongo等第三方Sink组件)

## 主要特性

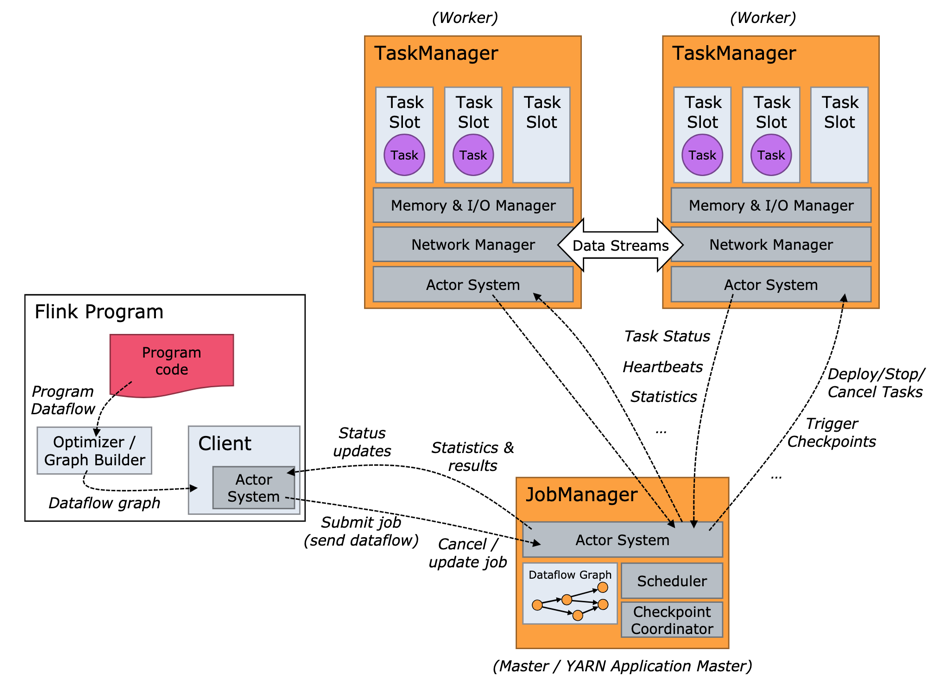
* 有界和无界数据流：支持处理无界和有界流
* 部署灵活：支持多种资源调度器，如YARN，kubernetes，以及自带的standalone调度器也足够灵活
* 极高可伸缩性：阿里双11，40亿/秒的处理速度，N个部署节点
* 极致流式处理性能：Flink相对Storm最大的特点就是将状态语义完全抽象到了框架中，支持本地状态读取（即读当前机器内存中的数据，而不是像Storm依赖redis等三方组件），避免了大量的网络IO，可以极大提升状态存取的性能。
* 如何理解Flink天然分布式计算？X个数据源->Y个算子节点->Z个输出目的地
* Flink内存管理：Flink 在 JVM 中提供了自己的内存管理，使其独立于 Java 的默认垃圾收集器。 它通过使用散列，索引，缓存和排序有效地进行内存管理。

## 运维、部署、监控

* 具备7\*24小时的高可用架构能力；一致性checkpoint(出故障准确恢复)，高效的checkpoint(快速恢复)
* Flink本身提供监控、运维的API、metrics，并有内置的webUI
* 部署模式：原则上只要TaskManager和JobManager只要能通信即可
  + 单机Standalone模式：本地单机跑，用于调试代码
  + 多机Standalone Cluster模式：Linux机器上搭建纯Flink集群
  + Yarn集群模式：依赖Hadoop，Flink跑在yarn之上，面向yarn分配资源，yarn面向linux申请资源。类似于k8s一样。
* Resource Manager
* Node Manager
* Yarn Container
  + Flink Job Manager –> AppMaster
  + Flink Task Manager -> AppWorker
* Yarn Session：通过session与yarn交互



* 环境：
  + Flink是基于Java(不低于1.8版本)和Scala作为开发语言，代码托管在github，并使用maven进行构建；可运行于Linux、MacOS、~~Windows~~系统
  + 部署试图



## API&核心类

* 数据源&执行环境：
  + DateSet：分布式批处理 -> ExecutionEnvironment
  + DataStream：分布式流处理(如：N多数据分布在不同的机器，共同组成一个DataStream) –> StreamExecutionEnvironment
* 分组 - groupBy()：用于DataSet批处理
* 分区 - keyBy(): 用于Datastream流处理。根据指定的key进行重新分区(hash，即根据key的hashcode进行取模运算)，即将N个数据源的数据进行重新划分，如key1-v1,v2,v3...到机器1，key2-v1,v2,v3...到机器2...，重新平衡数据。
  + keyBy(下标 -> 0)
  + keyBy(字段名称，支持嵌套 -> user.address.city)
  + keyBy(自定义，实现KeySelector接口)
* 映射-map()/flatMap()
  + MapFunction: A -> B
  + FlatMapFunction
  + RichMapFunction: A -> open kafka connection -> B -> close kafka connection
  + RichFlatMapFunction
  + CoMapFunction
  + CoFlatMapFunction
* 过滤 - filter()
* 归纳 - reduce()
* 算子间数据传递模式：
  + One-to-one streams: 保持元素的分区和顺序
  + Redistributing streams：改变了元素的分区
    - keyBy(key) -> 根据指定的key通过hash做分区
    - broadcast() -> 当前算子节点处理完后，向后面的每个算子节点做广播
    - rebalance() -> 重新随机分区，如随机发给后面的算子节点
* 累加器
  + Accumulator
    - SimpleAccumulator：常用简单场景累加、计数
* IntCounter
* IntMaximum
* IntMinimum
* DoubleCounter
* ...
* Histogram：柱状图
  + - MyCustomAccumulator：实现Accumulatoror或者继承SimpleAccumulator，来实现自定义的累加器
* DataStream API
  + Graph: 执行计划
    - StreamGraph implements Pipeline(老版本叫StreamPlan)
    - JobGraph
    - ExecutionGraph
    - 物理执行图
  + StreamExecutionEnvironment: 自动判断执行环
* Datasource
  + 基于文件
  + 基于Socket
  + 基于Collection
  + 基于自定义数据源
    - SourceFunction
    - ParallelSourceFunction
    - RichParallelSourceFunction
* Transformation（Operator、操作符、算子）：将一个或多个DataStream转换为新的DataStream
* DataSink: 结果输出
  + 内置的各种connector
  + 自定义Sink：实现SinkFunction或者RichSinkFunction
* xxx