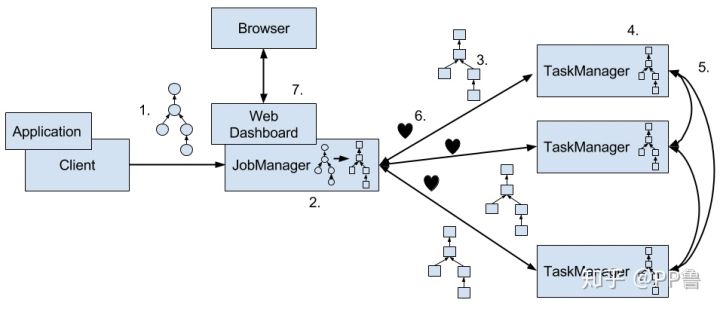
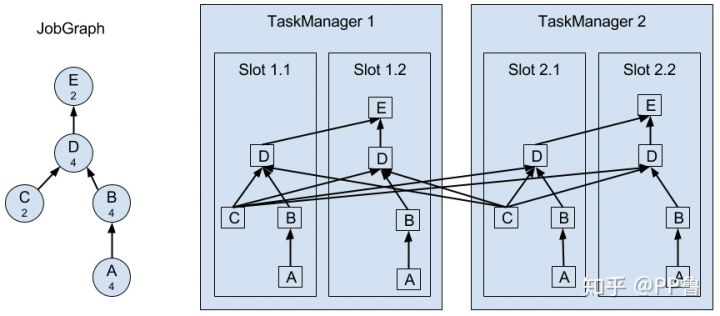
从知乎PP鲁的科技星球转载记录：

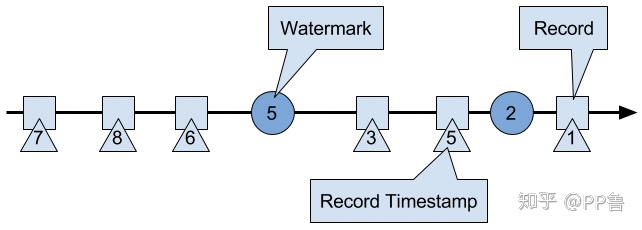


用户编写Flink应用并提交任务的具体流程为：

1. 用户在客户端（Client）编写应用程序代码。程序一般为Java或Scala语言，调用Flink API算子，构建基于逻辑视角的数据流图。代码和相关配置文件被编译打包，被提交到JobManager上，形成一个应用作业（Application）。
2. JobManager接受到作业后，将逻辑视角的数据流图转化成可并行执行的物理视角数据流图。
3. JobManager将物理视角数据流图发送给各TaskManager。
4. TaskManager执行被分配的任务。
5. TaskManager在执行任务过程中可能会与其他TaskManager交换数据。
6. TaskManager中的任务启动、运行、性能指标、结束或终止等状态信息会反馈给JobManager。
7. 用户可以使用Flink Web仪表盘来监控提交的作业。

上图展示了算子、任务以及槽位之间的关系：左侧为一个含有5个算子的逻辑视角数据流图，右侧为在TaskManager上执行的并行物理视角。Flink给这个作业分配2个TaskManager，每个TaskManager有2个槽位，共4个计算槽位。每个槽位都包含A、B、C、D算子子任务。A、B子任务在交换数据时不需要跨槽位，这将降低数据传输资源开销，C、D子任务之间会跨槽位，产生一些数据传输开销。

在实现TaskManager过程中，Flink在一个Java进程（Process）中启动多个线程（Thread）来并行执行这些任务。比起进程，线程的优势在于更轻量化、数据传输开销更小；线程的劣势是隔离性差，某一个任务出现错误可能导致整个TaskManager上的所有计算都崩溃。不过，Flink高度兼容不同的资源调度框架，如YARN、Mesos或Kubernetes，因此，为了有效隔离计算任务，可以给一个Flink任务单独创建一个Flink集群，或者在分配资源时将某台物理机上的所有资源都分配给同一个TaskManager，这样即使该应用出现问题，也不会影响其他应用。

Flink中的Watermark是被系统插入到数据流的特殊数据。Watermark的时间戳单调递增，且与事件时间戳相关。如上图的数据流所示，方块是事件，三角形是该事件对应的时间戳，圆圈为Watermark。当Flink接受到时间戳值为5的Watermark时，系统假设时间戳小于5的事件均已到达，后续到达的小于5的事件均为延迟数据。Flink处理到最新的Watermark，会开启这个时间窗口的计算，把这个Watermark之前的数据纳入进此次计算，延迟数据则不能被纳入进来，因此这种计算时有一定微小误差的。

延迟数据

Flink有一些机制专门收集和处理延迟数据。迟到事件在Watermark之后到达，一般处理的方式有三种：

1. 将迟到事件作为错误事件直接丢弃。
2. 将迟到事件收集起来另外再处理。
3. 重新触发计算。

对于第二种方式，用户可以使用Flink提供的Side Output机制，将迟到事件放入一个单独的数据流，以便再对其单独处理。

对于第三种方式，用户可以使用Flink提供的Allowed Lateness机制，设置一个允许的最大迟到时长，原定的时间窗口关闭后，Flink仍然会保存该窗口的状态，直至超过迟到时长，迟到的事件加上原来的事件一起重新被计算。

状态与检查点：

状态可以是当前所处理事件的位置偏移（Offeset）、一个时间窗口内的某种输入数据、或与具体作业有关的自定义变量。(还是不懂)。

Flink使用检查点（Checkpoint）技术来做失败恢复。检查点一般是将状态数据生成快照（Snapshot），持久化存储起来，一旦发生意外，Flink主动重启应用，并从最近的快照中恢复，再继续处理新流入数据。

Flink采用的是一种一致性检查点（Consistent Checkpoint）技术，它可以将分布在多台机器上的所有状态都记录下来，并提供了Exactly-Once的投递保障，其背后是使用了Chandy-Lamport算法，将本地的状态数据存储到一个存储空间上，并在故障恢复时在多台机器上恢复当前状态。

Flink提供了3中存储状态的方式： 内存、文件系统、RocksDB，统称为状态后端（State Backend）。

使用RocksDB作为后端时，Flink会将实时处理中的数据使用RocksDB存储在本地磁盘上。Checkpoint时，整个RocksDB数据库会被存储到配置的文件系统中，同时Flink会将极少的元数据存储在JobManager的内存中，或者在Zookeeper中（高可用情况）。

RocksDB支持增量Checkpoint，即只对修改的数据做备份，因此非常适合超大状态的场景。

源码中关于三种时间的解释：

ProcessingTime：

*/\*\*  
 \* Processing time for operators means that the operator uses the system clock of the machine  
 \* to determine the current time of the data stream. Processing-time windows trigger based  
 \* on wall-clock time and include whatever elements happen to have arrived at the operator at  
 \* that point in time.  
 \*  
 \* <p>Using processing time for window operations results in general in quite non-deterministic  
 \* results, because the contents of the windows depends on the speed in which elements arrive.  
 \* It is, however, the cheapest method of forming windows and the method that introduces the  
 \* least latency.  
 \*/*

IngestionTime:

*/\*\*  
 \* Ingestion time means that the time of each individual element in the stream is determined  
 \* when the element enters the Flink streaming data flow. Operations like windows group the  
 \* elements based on that time, meaning that processing speed within the streaming dataflow  
 \* does not affect windowing, but only the speed at which sources receive elements.  
 \*  
 \* <p>Ingestion time is often a good compromise between processing time and event time.  
 \* It does not need and special manual form of watermark generation, and events are typically  
 \* not too much out-or-order when they arrive at operators; in fact, out-of-orderness can  
 \* only be introduced by streaming shuffles or split/join/union operations. The fact that  
 \* elements are not very much out-of-order means that the latency increase is moderate,  
 \* compared to event  
 \* time.  
 \*/*

EventTime：

*/\*\*  
 \* Event time means that the time of each individual element in the stream (also called event)  
 \* is determined by the event's individual custom timestamp. These timestamps either exist in  
 \* the elements from before they entered the Flink streaming dataflow, or are user-assigned at  
 \* the sources. The big implication of this is that it allows for elements to arrive in the  
 \* sources and in all operators out of order, meaning that elements with earlier timestamps may  
 \* arrive after elements with later timestamps.  
 \*  
 \* <p>Operators that window or order data with respect to event time must buffer data until they  
 \* can be sure that all timestamps for a certain time interval have been received. This is  
 \* handled by the so called "time watermarks".  
 \*  
 \* <p>Operations based on event time are very predictable - the result of windowing operations  
 \* is typically identical no matter when the window is executed and how fast the streams  
 \* operate. At the same time, the buffering and tracking of event time is also costlier than  
 \* operating with processing time, and typically also introduces more latency. The amount of  
 \* extra cost depends mostly on how much out of order the elements arrive, i.e., how long the  
 \* time span between the arrival of early and late elements is. With respect to the  
 \* "time watermarks", this means that the cost typically depends on how early or late the  
 \* watermarks can be generated for their timestamp.  
 \*  
 \* <p>In relation to {****@link*** *#IngestionTime}, the event time is similar, but refers the the  
 \* event's original time, rather than the time assigned at the data source. Practically, that  
 \* means that event time has generally more meaning, but also that it takes longer to determine  
 \* that all elements for a certain time have arrived.  
 \*/*