Apache Flink的特性、概念和架构

可以将数据处理可以抽象成两方面的内容：数据集类型(types of datasets)和数据据集的处理模型(Execution Models)：

1）数据集类型，分为两大类：Unbounded（无限数据集），快速持续到达的流式数据，例如传感器实时测量数据、日志等；Bounded(有限数据集)，通过不可改变，不会发生更新的数据集。

2）数据处理模型，与数据集类型相对应，处理模型分为两类：Streaming（流式处理），在数据不断产生的同时持续处理数据；Batch（批式处理），在有限的时间内完成一批数据完整的处理，在处理后释放计算资源。

在数据处理模型（数据引擎）的发展上去看，最早出现的计算引擎是Hadoop MapReduce，其属于批式处理，其将计算分成两个阶段：Map和Reduce，对于上层应用来说，需要拆分算法甚至实现多个Job的串联来完成迭代计算，性能较低，为了解决这个问题产生了Tez及更上层的Oozie等DAG框架。

但是MR针对批处理，计算性能不高，以Spark为代表的内存计算，通过在Job内部支持DAG一定程度上达到准实时计算，但是本质上说Spark还是处理的有限数据集，Spark Streaming对Spark API进行扩展，实现高吞吐量、具备容错机制的实时流数据处理，但是其底层实现本质是批处理模型，其对流式数据进行离散化处理，将流式数据抽象成以时间片为单位的批数据，由于其原理模型的限制，其仅能达到一个准实时的处理效果。

为了支持流式数据（Unbounded）的处理，进一步提升实时性，出现了Apache Storm、Samza等流处理引擎（实时计算），可以实时的接收每条消息并顺序的处理每一个请求。

Apache Flink是基于流式处理模型设计的，可以连续地处理不断产生的数据，但是Flink支持批数据的处理，其是流数据处理的一个极限特例。Flink可以支持本地的快速迭代，同时可以定制化内存管理。如果对比Flink和Spark，Flink并没有将内存完全交给应用层。就其框架本身与应用场景来说，Flink更相似于Storm，Flink的使用结构图如下所示：

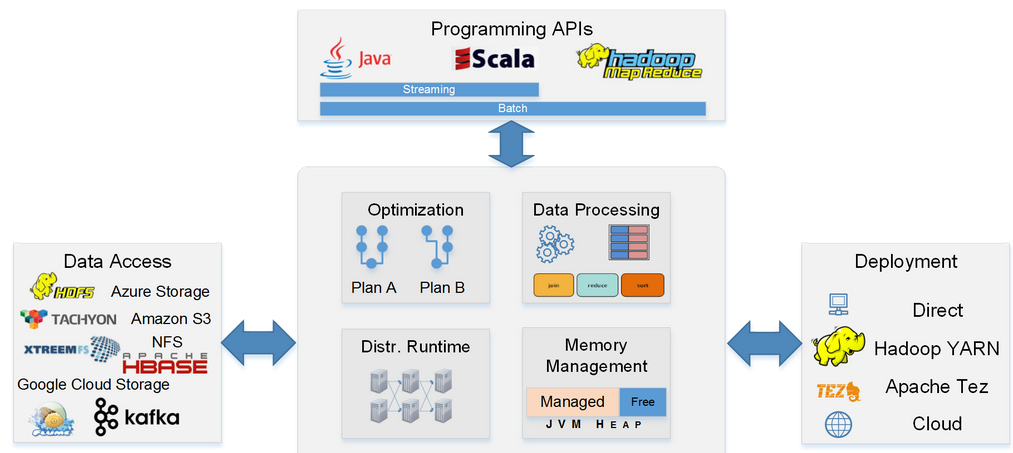


图1 .Flink生态图

其工作流程是首先从各种数据源收集事件，可以来自数据库、机器日志甚至事件传感器，这些事件需要清理组织化到一个中心集成各种数据流，典型的工作是Apache Kafka，Kafka提供一个Broker功能，以失败容错的高可靠性来收集日志或缓冲数据，以及分发到对不同流感兴趣的消费者进行分析。对流计算真正的分析，如创建计数器实现聚合、Map/Reduce类计算，将各种流Join一起分析等，Flink正是这步的实现。Apache Flink具有如下特性：

* 支持高吞吐、低延迟、高性能的流处理
* 支持带有事件时间的窗口操作
* 支持有状态的计算Exactly-once语义
* 支持高度灵活的窗口操作，支持基于time、count、session以及data-driven的窗口操作
* 支持具有轻量级分布式快照（Snapshot）实现的容错
* 一个运行时支持Batch on Streaming处理
* Flink在JVM内部实现了自己的内存管理
* 支持迭代计算
* 支持程序的自动优化，避免特定情况下Shuffle、排序等消耗时间的操作，中间结果进行缓存

Apache Flink在大数据生态圈中有一个完整的Stack，能够和其他大数据处理项目和框架无缝结合，如下图所示：

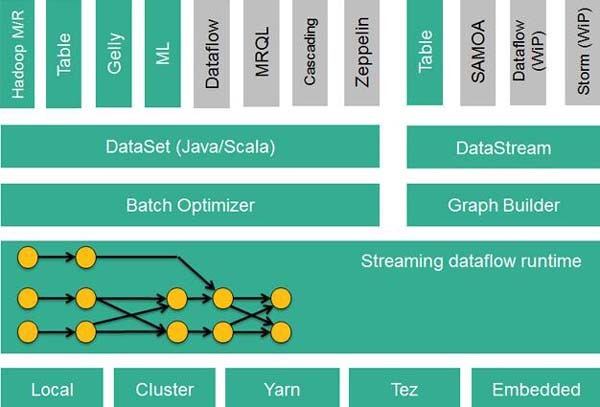


图2. Flink Stack

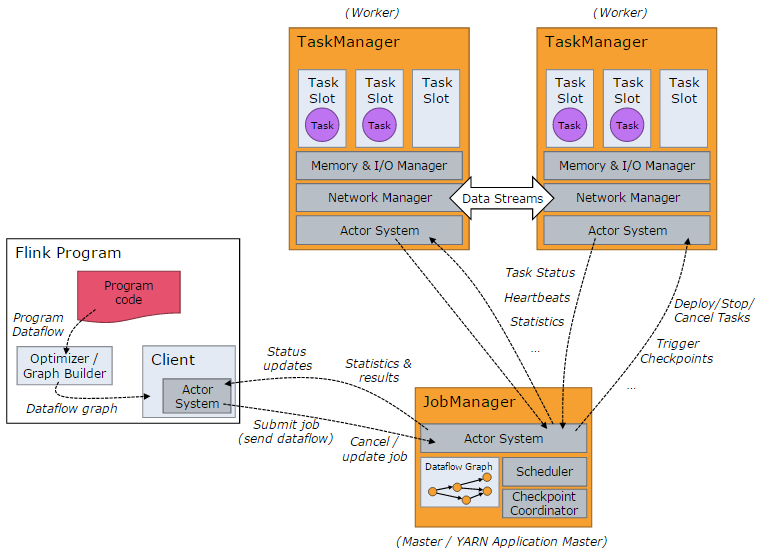
Flink实现了很多Connector的子项目，如与Hadoop HDFS集成，也支持Alluxio、S3、MapRFS等，目前支持的系统列表如下：

* Apache Kafka => sink/source
* Elasticsearch => sink
* HDFS => sink
* RabbitMQ => sink/source
* Amazon Kinesis Streams => sink/source
* Twitter => source
* Apache NIFI => sink/source
* Apache Cassandra => sink
* Redis, Flume,ActiveMQ => sink

可以使用上述的connetors来运行应用，目前在社区中有很多基于Flink构建的第三方包，如Apache Zeppelin,Apache Beam等使用Flink来做执行引擎。

# Apache Flink系统架构

Apache Flink由Java代码实现，在Flink中会把所有的任务当做流来处理，其系统架构与Spark类似，基于Master-Slave的架构，如下图所示：



Flink集群启动后，会启动一个JobManager和多个TaskManager进程，在Local模式下会在同一个JVM内部启动一个JobManager和TaskManager进程。当Flink程序提交后，会创建一个Client进行预处理，并转化成一个并行数据流，对应为Flink Job，从而可以被JobManager和TaskManager执行。Flink基于actor实现JobManager和TaskManager，组件之间的信息交换都是通过事件的方式进行。Flink包括如下3个主要的进程：

**1）JobManager**

Flink的协调者，负责接收Flink Job，调度组成Job的多个Task的执行，同时JobManager还负责收集Job的状态信息，并管理Flink集群中从节点TaskManager。JobManager所负责的各项管理功能，接收并处理的事件包括：

* RegisterTaskManager，在Flink集群启动的时候，TaskManager会向JobManager注册，如果注册成功，则JobManager会向TaskManager回复消息AcknowledgeRegistration。
* SubmitJob，Flink程序内部通过Client向JobManager提交Flink Job，其中在消息SubmitJob中以JobGraph的形式描述Job的基本信息
* CancelJob，请求取消一个Flink Job的执行，CancelJob消息中包含Job的ID，如果成功则返回消息CancellationSuccess，失败则返回消息CancellationFailure
* UpdateTaskExcecutionState，TaskManager会向JobManager请求更新ExecutionGraph中的ExecutionVertex的状态信息，更新成功则返回True
* RequestNextInputSplit，运行在Task之上的Task，请求获取下一个要处理的输入Split成功则返回NextInputSplit
* JobStatusChanged，ExectionGraph向JobManager发送该消息，用来表示Flink Job的状态发生的变化，例如RUNNING,CANCELLING及FINISHED等

**2）TaskManager**

TaskManager也是一个Actor，它是实际负责执行计算的Worker，在其上执行Flink Job的一组Task。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。TaskManager端可以分成两个阶段：

* 注册，向JobManager注册，发送RegisterTaskManager消息，等待JobManager返回AcknowledeRegistration，然后TaskManager就可以进行初始化过程。
* 可操作阶段，该阶段TaskManager可以接收并处理与Task有关的消息，如SubmitTask、CancelTask、FailTask。如果TaskManager无法连接到JobManager，则TaskManager失去与JobManager的联系，会自动进入注册阶段，只有完成注册才能继续处理Task相关的消息。

**3）Client**

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地址，并建立与JobManager的连接，将Flink Job提交给JobManager。

Client会将用户提交的Flink程序组一,JobGraph，并以JobGraph的形式提交。一个JobGraph是一个Flink Dataflow，它由多个JobVertex组成的DAG。其中，一个JobGraph包含了一个Flink程序的如下信息：JobID、Job名称、配置信息和一组JobVertex组成。

客户端不是运行时和程序执行的一部分，其用于准备发送dataflow给master，然后客户端断开链接或者维持连接以等待接收计算结果。客户端以两种方式运行：要么作为Java/Scala程序的一部分被触发执行，要么以命令行flink run的方式执行。

# Apache Flink基本概念

Flink的核心是一个流式的数据流执行引擎，其针对数据流的分布式计算提供了数据分布、数据通信以及容错机制等功能。基于流执行引擎，Flink提供了诸多更高抽象层的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数据集，用户可方便的使用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，通过类SQL的DSL对关系表进行各种查询操作，支持Java和Scala

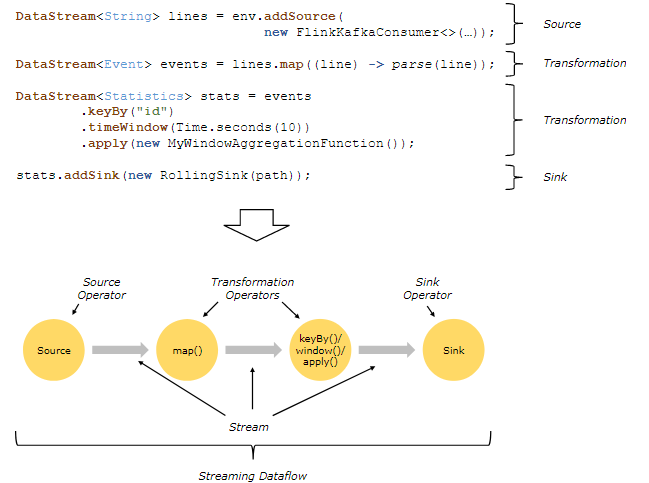
Flink还针对特定的应用领域提供领域库，如机器学习库：Flink ML，图计算库：Gelly等。下面介绍Flink的基本概念。

## 2.1 Stream&Transformation&Operator

Apache Flink程序构建的基本单元是stream和transformation（DataSet实质上是stream）。Streams可以看做是中间结果，Transformations以一个或者多个stream作为输入的某种操作，该操作使用这些stream进行计算从而产生一个或多个Result Stream。在运行时，Flink上运行的程序会被映射为stream dataflows，其包括streams和transformations操作。

dataflow以sources开始，以一个或多个sinks结束，其类似于任意的有向无环图（DAG），特定形式的环可以通过iteration构建，在大多数情况下程序中transformations和dataflow中的operator是一一对应的关系，在某些情况下transformation可以对应多个operator。

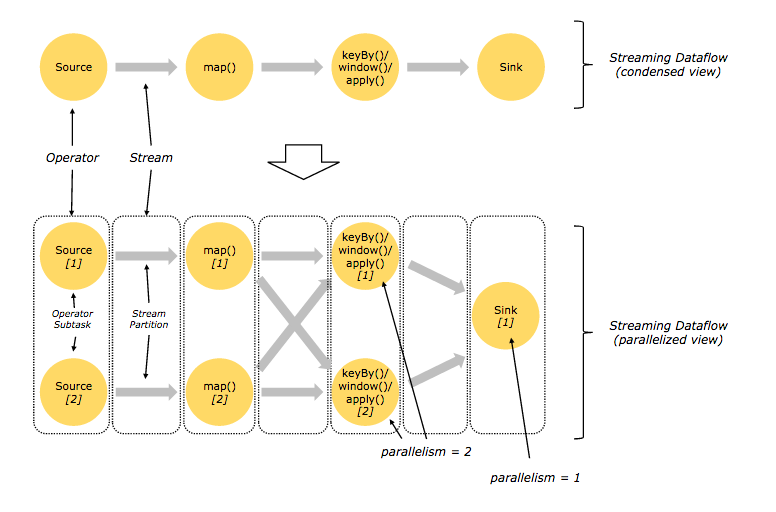
示例如下：



## 2.2 Parallel Dataflows

程序在Flink内部的执行具有并行、分布式的特性。stream被分割成stream partition，operator被分割成operator subtask，这些operator subtask在不同的线程、不同的物理机或不同的容器中彼此不依赖得执行。

特定的operator的subtask个数称之为parallelism(并行度)，一个stream的并行度总是等同于其producing operator的并行度，一个程序中不同的operator可能具有不同的并行度，如下图所示：

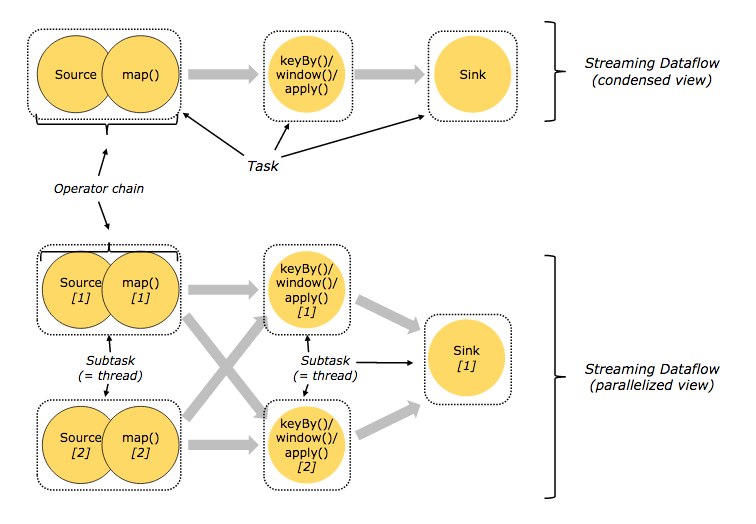


Stream在operator之间传输数据的形式可以是one-to-one(forwarding)的模式也可以是redistributing的模式：

* one-to-one: 比如在source和map 之间维护着分区以及元素的顺序，意味着map operator的subtask看到的元素个数以及顺序跟source operator的subtask生成的元素个数、顺序相同。
* Redistributing: map与keyBy/window以及keyBy/window与sink之间stream的分区会发生改变。每一个operator task依据所选择的transformation发送数据到不同的目标subtask。例如keyBy() 基于hash进行重分区，broadcase或者rebalance进行随机重分区。在一个redistribution的交换中，只有每个发送-接收的task对的顺序才会被维持，比如map的subtask和keyby/window的subtask。

## 2.3 tasks&operator chains

出于分布式执行的目的，Flink将Operator的Subtask链接在一起形成Task，每一个task在一个线程中执行。将operators链接成的task是非常有效的优化，能减少线程之间的切换和基于缓存区的数据交换，在减少时延的同时提升吞吐量。链接的形式可以在编程API中进行指定，下图中展示了5个Subtask以5个并行的线程来执行：

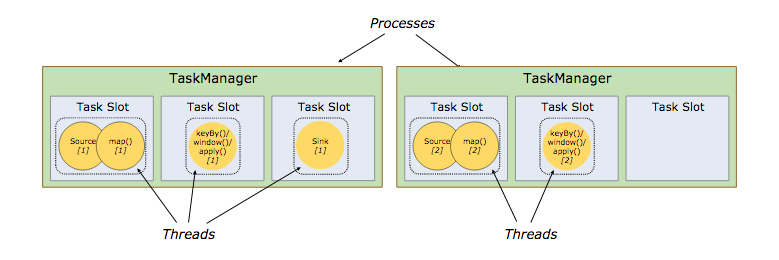


## 2.4 Workers、Slots及Resources

每一个Worker(TaskManager)是一个JVM进程，它可能会在独立的线程上执行一个或者多个subtask，为了控制一个worker能接收多少个task。Worker通过task slot来进行控制（一个worker至少有一个task slot）。

每个task slot表示TaskManager拥有资源的一个固定大小的子集，假如一个TaskManager有三个slot，会将其管理的内存分成三份给各个slot。资源slot化意味着一个subtask将不需要来自其他Job的Subtask竞争被管理的内存，取而代之的是它将拥有一定数量的内存储备。

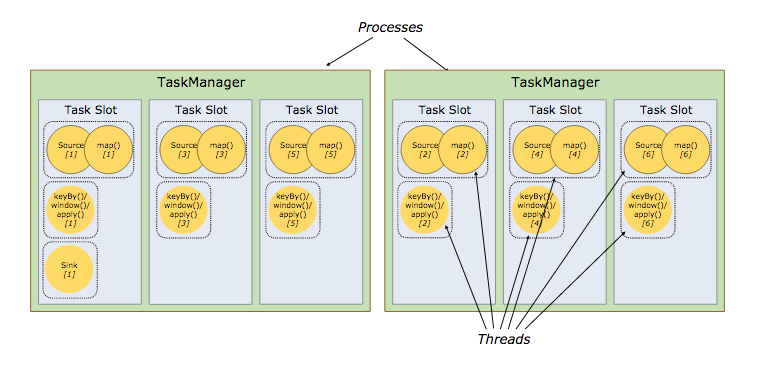
通过调整task slot的数量，允许用户定义subtask之间如何互相隔离，如果TaskManager启动一个Slot，那意味着每个task group运行在独立的JVM中。而一个TaskManager启动多个slot意味着多subtask可以共享同一个JVM。在同一个JVM进程中task将共享TCP连接（基于多路复用）和心跳信息。它们可能共享数据集和数据集，因此这减少了每个task的负载：



默认，如果subtask来自相同job，但不是相同task，Flink允许subtask共享task，结果是一个Slot可能hold住该job的整个pipeline，允许slot共享有以下两个好处：

* Flink集群确实需要许多task slots来让Job达到最大并行度，不需要计算一个程序总共包含多少个task
* 更容易获得更好的资源利用，如果没有slot共享，非密集型的source/map的subtask将阻塞跟密集型的window的subtask一样多的占用资源。而如果有slot共享，基本的并发度通过完整的利用共享的slot资源获得2到6倍的提升，同时仍然保证每一个TaskManager会在任务繁重的subtask之间合理的slot共享。

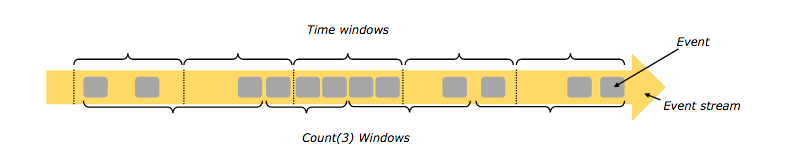
Slot共享行为可以通过API来控制，以防止不合理的共享，这个机制称之为resource groups，其定义了subtask可能共享的slot是什么资源。作为一个约定俗成的规则，task slot推荐的默认值是cpu合数，基于超线程技术，每个slot占用两个或更多的实际线程上下文：



## 2.5 窗口

聚合事件（比如count,sum）和批处理略微不同，例如不能一次完成多流中所有元素的数量统计，然后返回解耦。因为Stream数据通常是无限的，取而代之的是在流上的聚合（count,sum）被隔离到window域中，比如统计最近5分钟的数量或对最近100个元素求和。

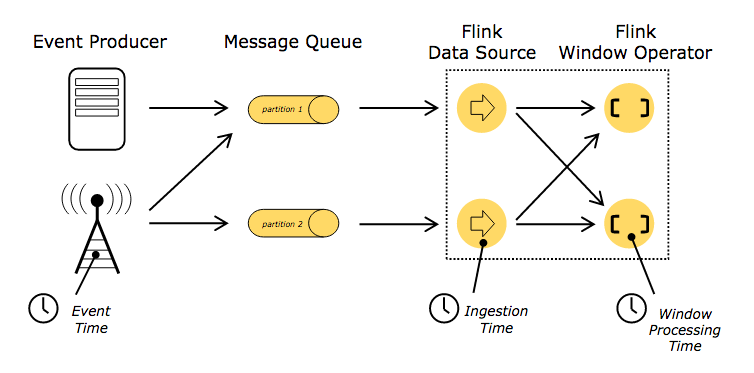
窗口可以是时间驱动（比如每30秒）也可以是数据驱动（比如，每100个元素）。通常将窗口划分为tumbing windows(不重叠)、sliding windows(有重叠)和session windows。



## 2.6 时间

当再流式编程中涉及到时间（比如定义一个窗口），可能涉及到时间的不同定义：

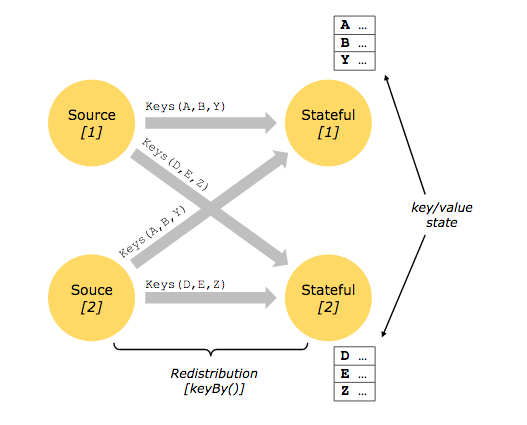
* Event Time，指一个事件的创建时间，通常在event中用时间戳来描述，比如可能是由生产事件的传感器或生产服务来附加。Flink访问事件时间戳通过时间戳分配器
* Ingestion time，指一个事件从source operator进入Flink Dataflow的时间
* Processing time，每一个执行一个基于时间操作的operator本地时间



## 2.7 状态和失败容忍

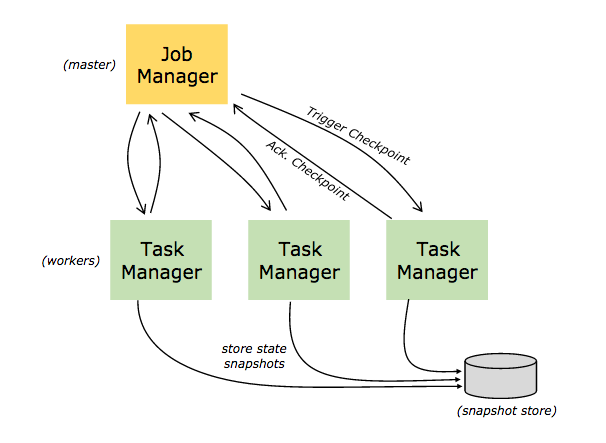
在dataflow中许多操作一次只关注一个独立事件（比如事件解析器），还有一些操作能记住多个独立事件的信息（比如window operator），而这些操作被称为stateful(有状态的)。

有状态的操作，其状态被维护的地方，可以将其看作是一个内嵌的key/value存储器。状态和流一起被严格得分区和分布以供有状态的operator读取。因此，访问key/value的状态仅能在**keyed streams**中（在执行keyBy()函数之后产生keyed stream），并且只能根据当前事件的键来访问其值。对齐stream的键和状态可以确保所有的状态更新都是本地操作，在不需要事务开销的情况下保证一致性。这个对齐机制也允许Flink重新分布状态并显式调整stream的分区



## 2.8 用于失败容忍的检查点

Flink实现失败容忍使用了流重放和检查点的混合机制。一个检查点会在流和状态中定义一个一致点，在该一致点streaming dataflow可以恢复并维持一致性（exactly-once的处理语义）。在最新的检查点之后的事件或状态更新将在input stream中被重放。检查点的设置间隔意味着在执行时对失败容忍产生的额外开销以及恢复时间（也决定了需要被重放的事件数）



## 2.9 状态的最终存储

给key/value构建索引的数据结构最终被存储的地方取决于状态最终存储的选择。其中一个选择是在内存中基于hash map，另一个是RocksDB。另外用来定义Hold住这些状态的数据结构，状态的最终存储也实现了基于时间点的快照机制，给key/value做快照，并将快照作为检查点的一部分来存储。

## 2.10 基于流的批处理

Flink执行批处理程序是将其作为流处理程序的一个特例来看待。它将其看作有界的流（有限数量的元素）。DataSet在内部被当作一个流数据，因此上面的这些适用于流处理的这些概念在批处理中同样适用，只有很少的几个例外：

* DataSet的编程API不适用检查点。恢复机制是通过重放完整的流数据来进行。那是合理的，因为输入时有界的。它将开销更多地引入到恢复操作上，但另一方面也使得运行时的常规流程代价更低，因为它规避了检查点机制。
* DataSet的有状态的operation API简单地使用in-memory/out-of-core的数据结构，而不是基于key/value的索引机制
* DataSet的API引进了独特的同步迭代机制（基于superstep），它仅在有界的流中存在。

# Apache Flink的部署方式

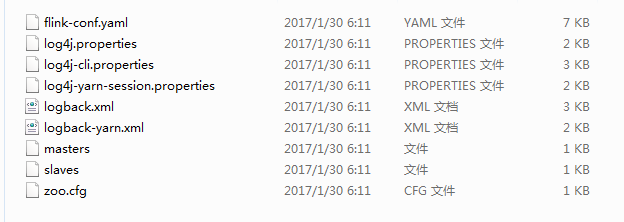
Flink有三种部署方式，分别是Local，Standalone Cluster和Yarn Cluster

## Local模式

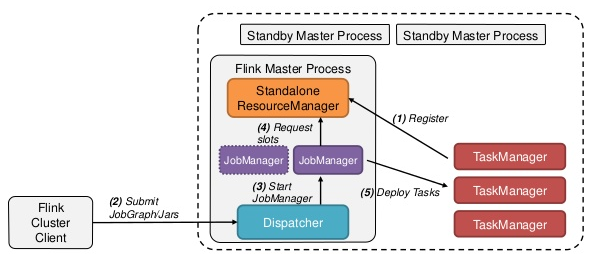
Local模式下，JobManager和TaskManager会共用一个JVM来完成Workload。如果要验证一个简单的应用，Local模式是最方便的，实际上应用中大多使用Standalone或者YARN Cluster。但是实际应用上大多使用Standalone或者YARN Cluster。

## Standalone模式

在搭建Standalone模式的Flink集群之前，首先下载Flink安装包，下载并解压后，进入到Flink根目录，然后查看conf文件夹，如下图所示：

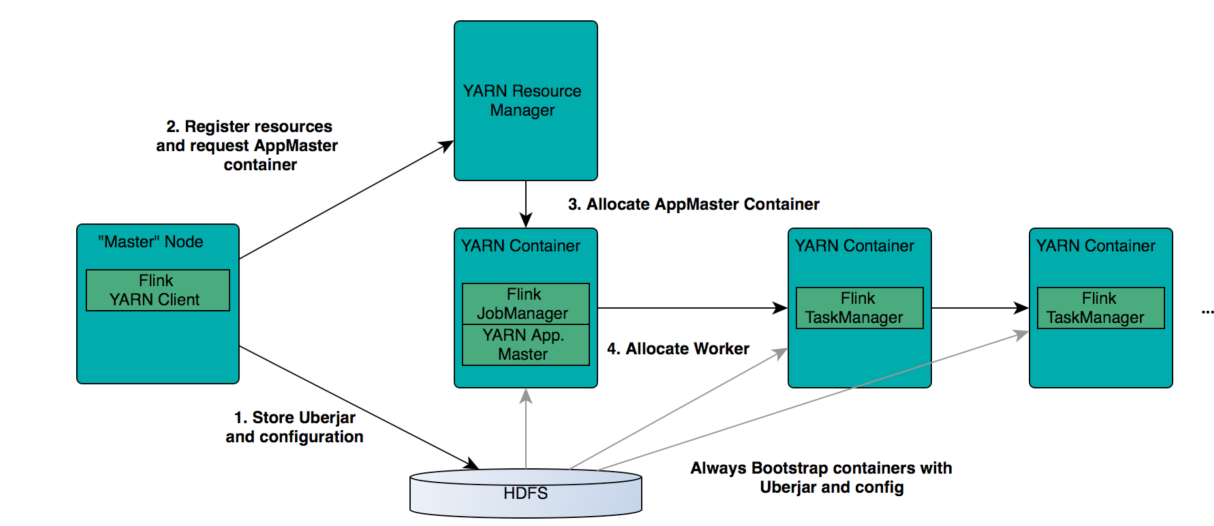


需要指定Master和Worker，Master会启动JobManager，Worker则会启动TaskManager，因此需要修改conf目录中的master和slaves。在配置master文件时，需要指定JobManager的UI监听端口。一般情况下，JobManager只需配置一个，Worker则配置一个或多个



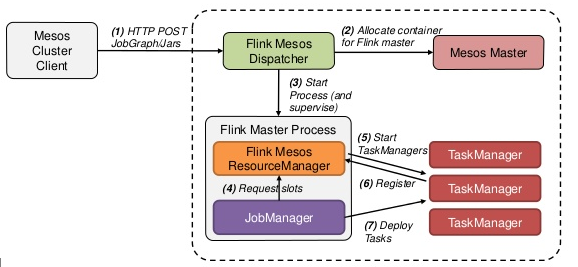
## YARN Cluster模式

Flink部署在YARN上，仅作为YARN上的多租户的一个service而存在。Flink与YARN 之间的关系如下图所示：



从图中可以看出，Flink与YARN的关系与MapReduce和YARN的关系是一样的。Flink通过YARN的接口实现了自己的App Master。当在YARN中部署了Flink，YARN就会用自己的Container来启动Flink的JobManager和TaskManager。

将Flink部署在Mesos中的方式与YARN 类似，不再详细介绍，系统结构图如下所示：



Mesos Cluster

# 参考文献

https://blog.csdn.net/yanghua\_kobe/article/details/51298871

https://ci.apache.org/projects/flink/flink-docs-release-1.0/concepts/concepts.html

https://blog.csdn.net/lmalds/article/details/60575205

http://bigdata.51cto.com/art/201611/522441.htm

https://blog.csdn.net/lmalds/article/details/60575205

http://www.uml.org.cn/yunjisuan/201610264.asp?artid=18570

https://www.ibm.com/developerworks/cn/opensource/os-cn-apache-flink/