# Flink，Storm，Spark Streaming三种流框架的对比分析

Author：yuemeng

Company：NetEase

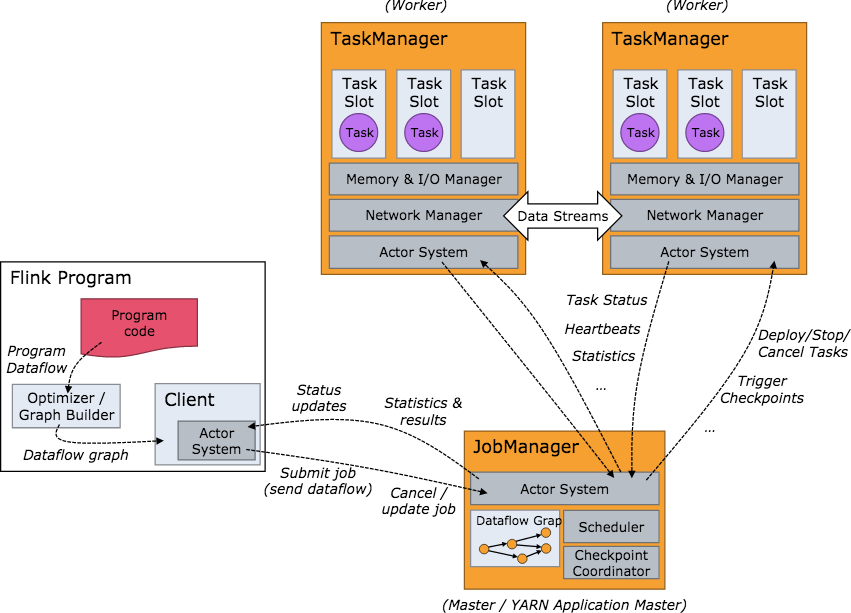
Date： 2016/08/05

## 1.Flink架构及特性分析

Flink是个相当早的项目，开始于2008年，但只在最近才得到注意。Flink是原生的流处理系统，提供high level的API。Flink也提供API来像Spark一样进行批处理，但两者处理的基础是完全不同的。Flink把批处理当作流处理中的一种特殊情况。在Flink中，所有的数据都看作流，是一种很好的抽象，因为这更接近于现实世界。

## 1.1 基本架构

下面我们介绍下Flink的基本架构，Flink系统的架构与Spark类似，是一个基于Master-Slave风格的架构。



当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。

Client 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。

JobManager 主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。

TaskManager 在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

可以看到 Flink 的任务调度是多线程模型，并且不同Job/Task混合在一个 TaskManager 进程中。主要组件为JobManager，TaskManager，Client,下面简要描述这三个组件的作用。

### JobManager

JobManager是Flink系统的协调者，它负责接收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，它接收到并处理的事件主要包括：

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形式描述了Job的基本信息。

CancelJob

请求取消一个Flink Job的执行，CancelJob消息中包含了Job的ID，如果成功则返回消息CancellationSuccess，失败则返回消息CancellationFailure。

UpdateTaskExecutionState

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TaskManager上面的Task，请求获取下一个要处理的输入Split，成功则返回NextInputSplit。

JobStatusChanged

ExecutionGraph向JobManager发送该消息，用来表示Flink Job的状态发生的变化，例如：RUNNING、CANCELING、FINISHED等。

### 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。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。TaskManager端可以分成两个阶段：

注册阶段

TaskManager会向JobManager注册，发送RegisterTaskManager消息，等待JobManager返回AcknowledgeRegistration，然后TaskManager就可以进行初始化过程。

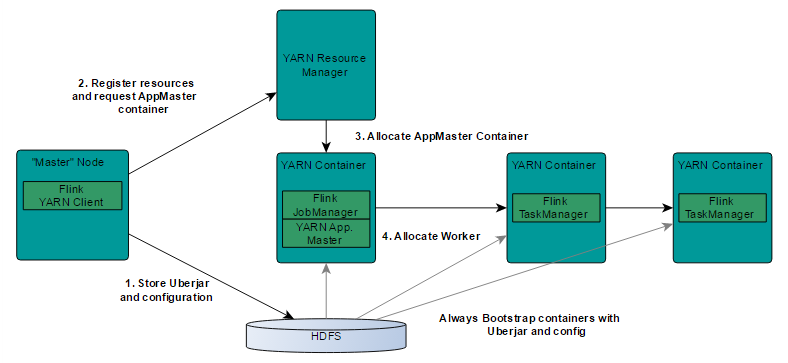
可操作阶段

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，这是TaskManager就失去了与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。

### 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。Client会将用户提交的Flink程序组装一个JobGraph， 并且是以JobGraph的形式提交的。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、一组JobVertex等。

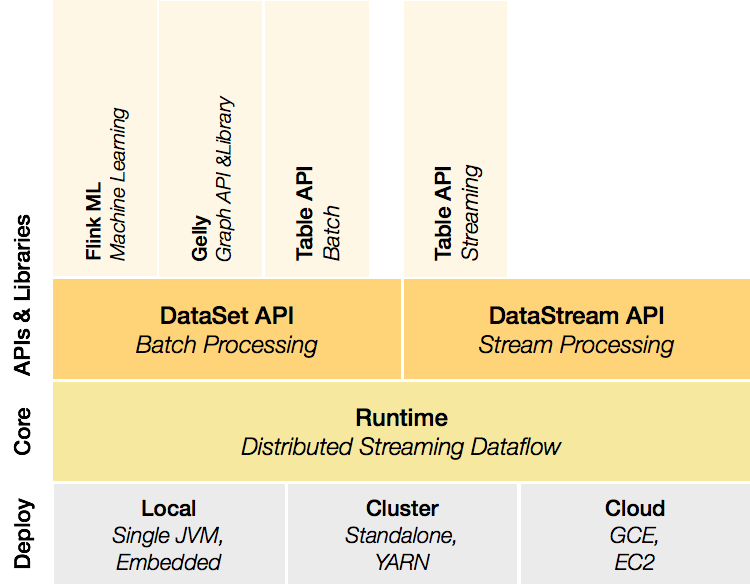
## 1.2 基于Yarn层面的架构



基于yarn层面的架构类似spark on yarn模式，都是由Client提交App到RM上面去运行，然后RM分配第一个container去运行AM，然后由AM去负责资源的监督和管理。需要说明的是，Flink的yarn模式更加类似spark on yarn的cluster模式，在cluster模式中，dirver将作为AM中的一个线程去运行，在Flink on yarn模式也是会将JobManager启动在container里面，去做个driver类似的task调度和分配，YARN AM与Flink JobManager在同一个Container中，这样AM可以知道Flink JobManager的地址，从而AM可以申请Container去启动Flink TaskManager。待Flink成功运行在YARN集群上，Flink YARN Client就可以提交Flink Job到Flink JobManager，并进行后续的映射、调度和计算处理。

## 1.3 组件栈

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。



### Deployment层

该层主要涉及了Flink的部署模式，Flink支持多种部署模式：本地、集群（Standalone/YARN）、云（GCE/EC2）。Standalone部署模式与Spark类似，这里，我们看一下Flink on YARN的部署模式

### Runtime层

Runtime层提供了支持Flink计算的全部核心实现，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。

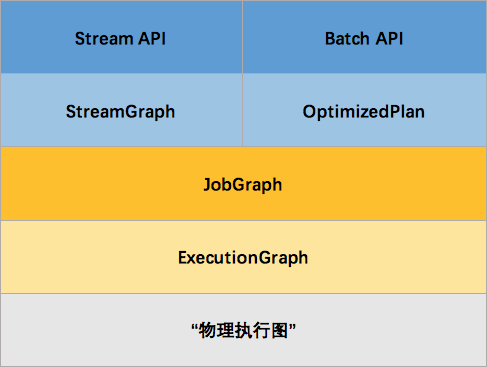
### API层

API层主要实现了面向无界Stream的流处理和面向Batch的批处理API，其中面向流处理对应DataStream API，面向批处理对应DataSet API。

### Libraries层

该层也可以称为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）。

从官网中我们可以看到，对于Flink一个最重要的设计就是Batch和Streaming共同使用同一个处理引擎，批处理应用可以以一种特殊的流处理应用高效地运行。

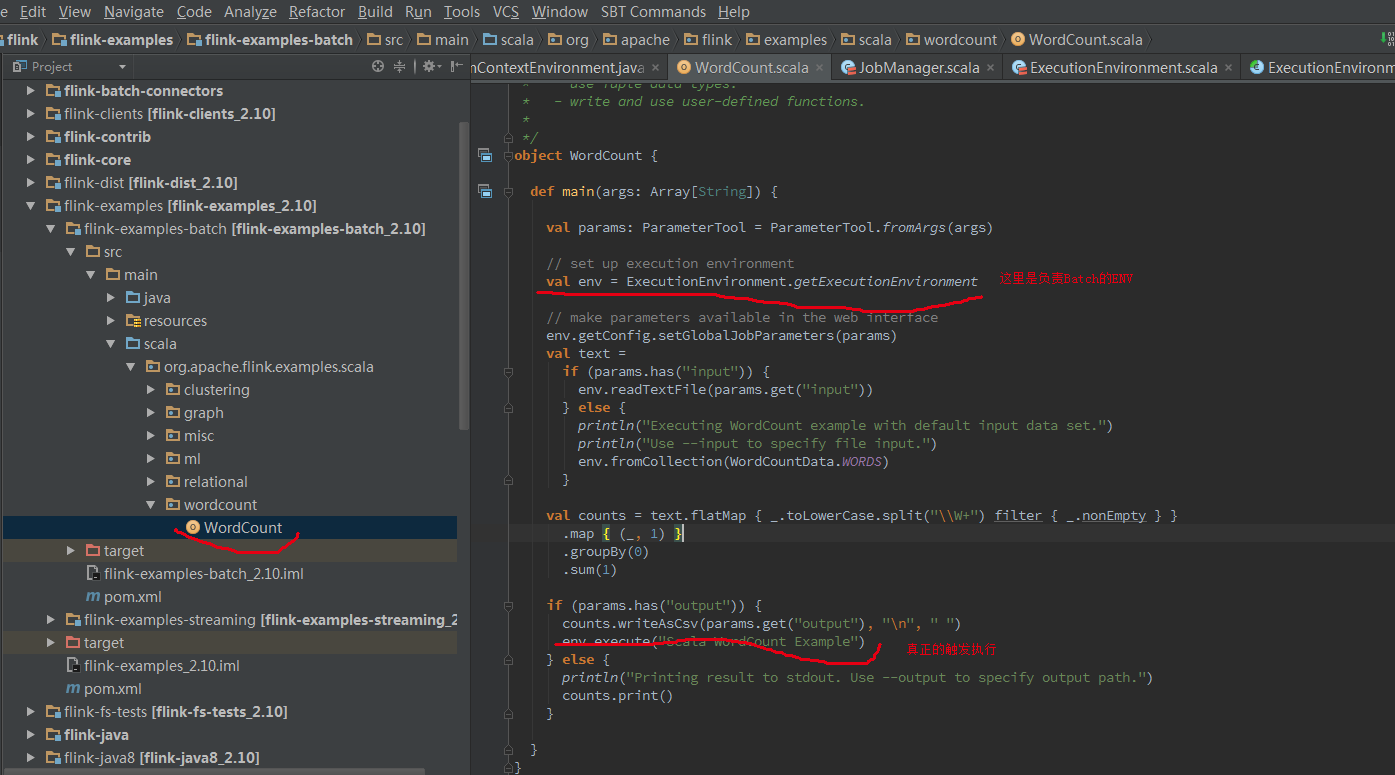


这里面会有一个问题，就是Batch和Streaming是如何使用同一个处理引擎进行处理的。

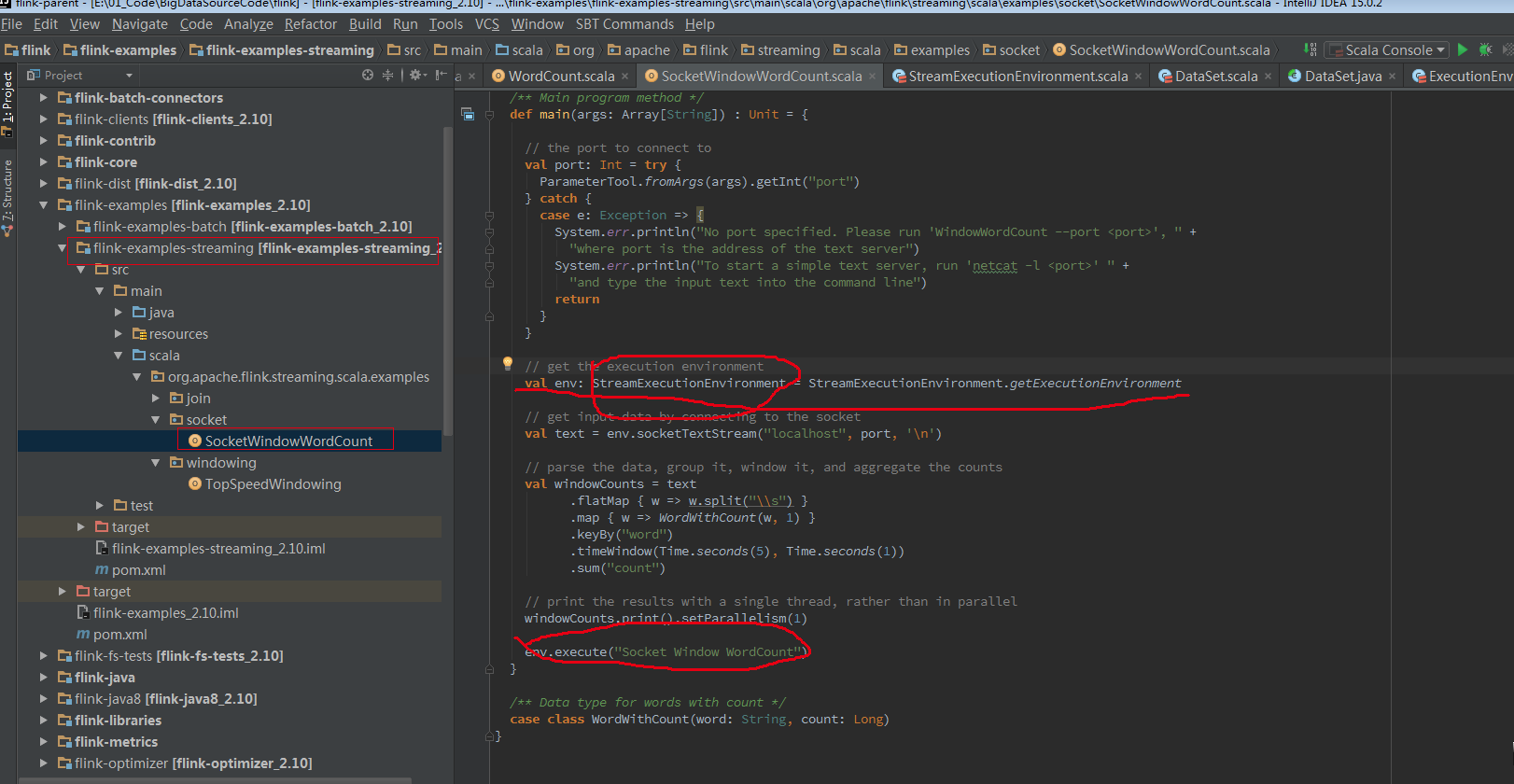
## 1.4 Batch和Streaming是如何使用同一个处理引擎。

下面将从代码的角度去解释Batch和Streaming是如何使用同一处理引擎的。首先从Flink测试用例来区分两者的区别。

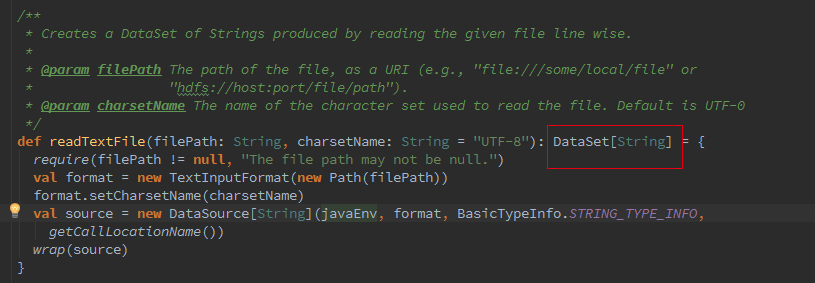
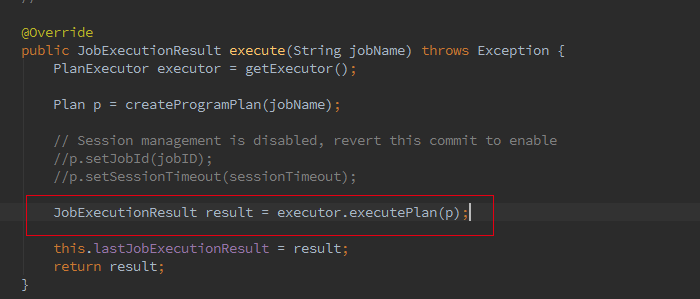
Batch WordCount Examples

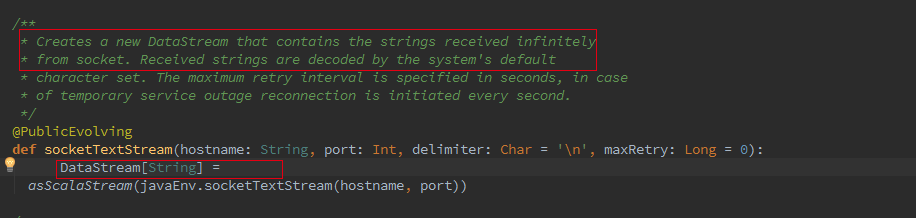


Streaming WordCount Examples

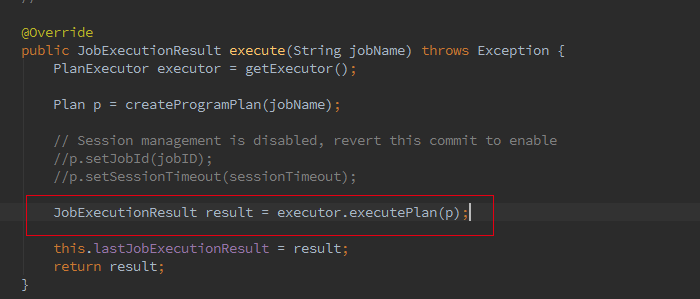


Batch和Streaming采用的不同的ExecutionEnviroment，对于ExecutionEnviroment来说读到的源数据是一个DataSet,而StreamExecutionEnviroment的源数据来说则是一个DataStream。

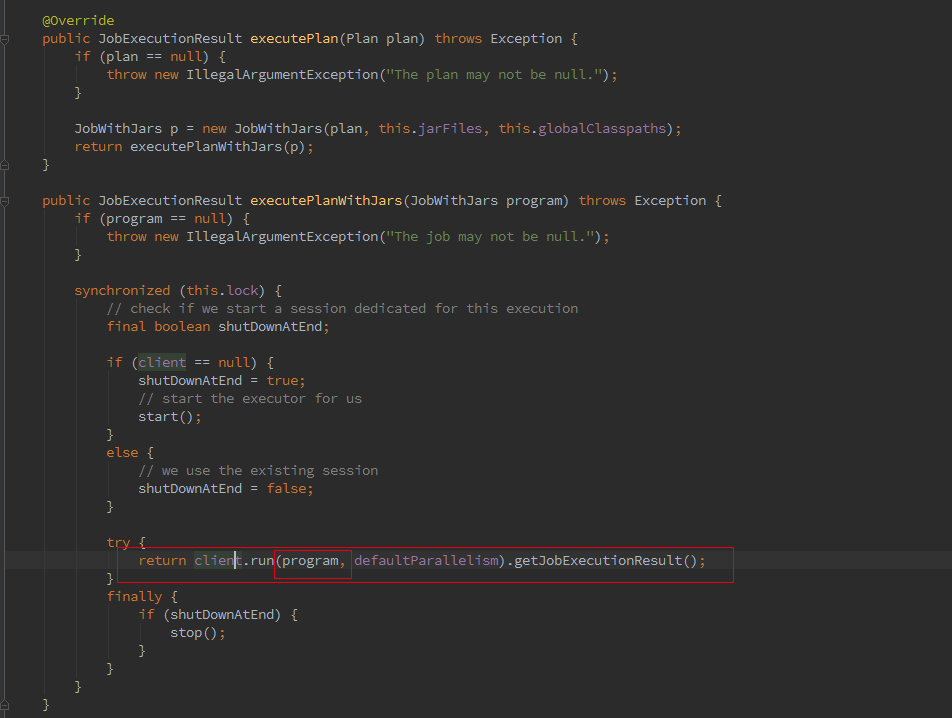


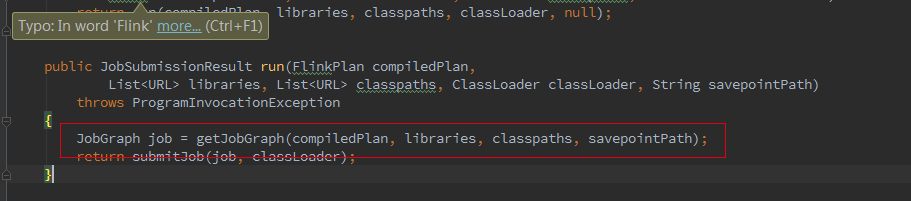


接着我们追踪下Batch的从Optimzer到JobGgraph的流程，这里如果是Local模式构造的是LocalPlanExecutor，这里我们只介绍Remote模式，此处的executor为RemotePlanExecutor

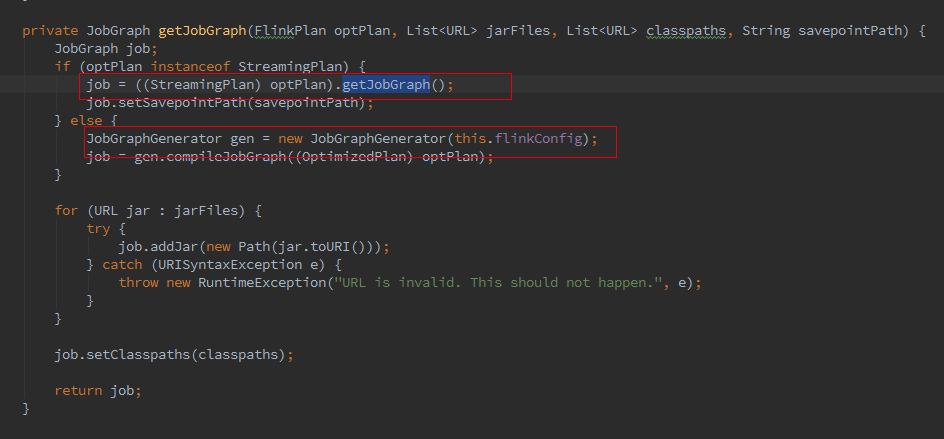


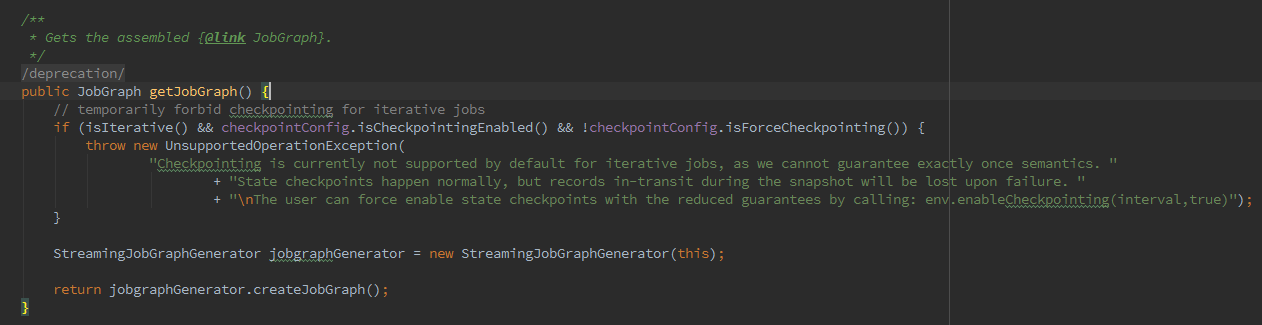
最终会调用ClusterClient的run方法将我们的应用提交上去，run方法的第一步就是获取jobGraph，这个是client端的操作，client会将jobGraph提交给JobManager转化为ExecutionGraph。Batch和streaming不同之处就是在获取JobGraph上面。





如果我们初始化的FlinkPlan是StreamingPlan，则首先构造Streaming的StreamingJobGraphGenerator去将optPlan转为JobGraph，Batch则直接采用另一种的转化方式。



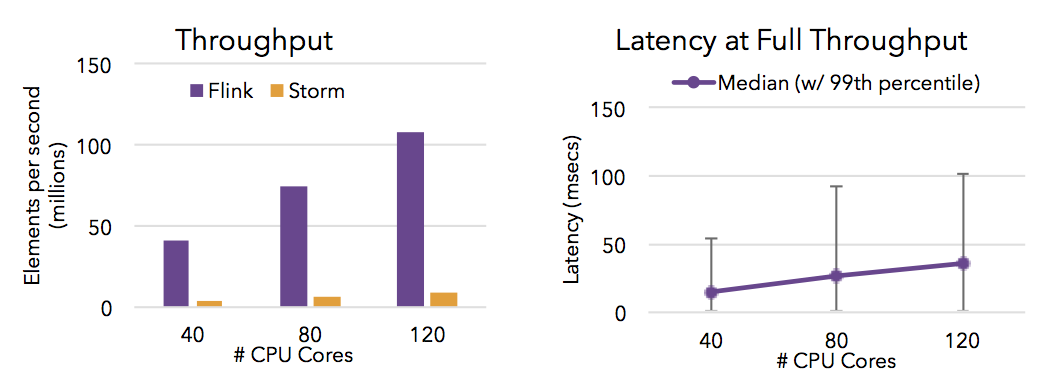


简而言之，Batch和streaming会有两个不同的ExecutionEnvironment，不同的ExecutionEnvironment会将不同的API翻译成不同的JobGgrah，JobGraph 之上除了 StreamGraph 还有 OptimizedPlan。OptimizedPlan 是由 Batch API 转换而来的。StreamGraph 是由 Stream API 转换而来的，JobGraph 的责任就是统一 Batch 和 Stream 的图。

## 1.5 特性分析

### 高吞吐 & 低延迟

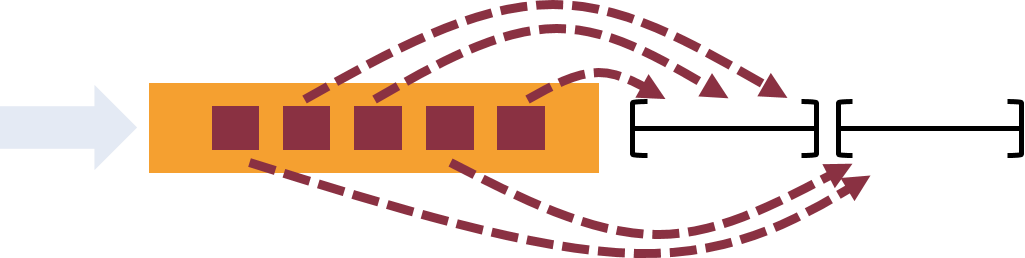
Flink 的流处理引擎只需要很少配置就能实现高吞吐率和低延迟。下图展示了一个分布式计数的任务的性能，包括了流数据 shuffle 过程。



### 支持 Event Time 和乱序事件

Flink 支持了流处理和 **Event Time** 语义的窗口机制。

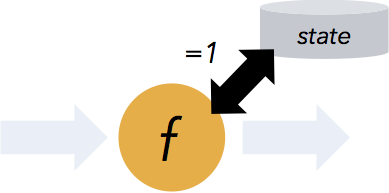
Event time 使得计算乱序到达的事件或可能延迟到达的事件更加简单。



### 状态计算的 exactly-once 语义

流程序可以在计算过程中维护自定义状态。

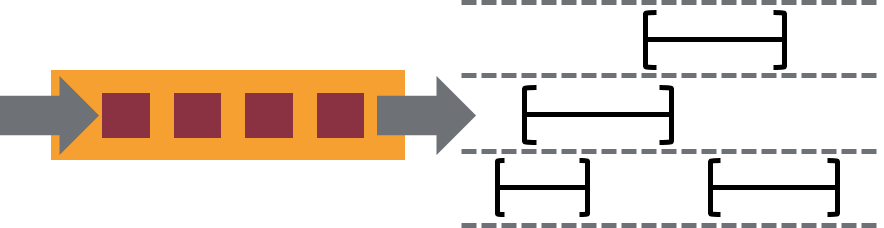
Flink 的 checkpointing 机制保证了即时在故障发生下也能保障状态的 *exactly once* 语义。



### 高度灵活的流式窗口

Flink 支持在时间窗口，统计窗口，session 窗口，以及数据驱动的窗口

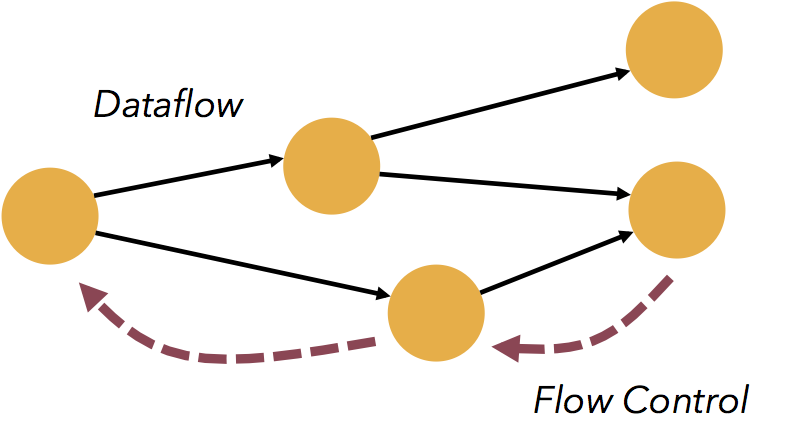
窗口可以通过灵活的触发条件来定制，以支持复杂的流计算模式。



### 带反压的连续流模型

数据流应用执行的是不间断的（常驻）operators。

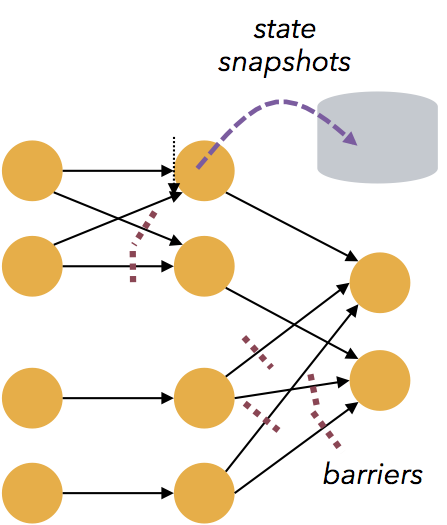
Flink streaming 在运行时有着天然的流控：慢的数据 sink 节点会反压（backpressure）快的数据源（sources）。



### 容错性

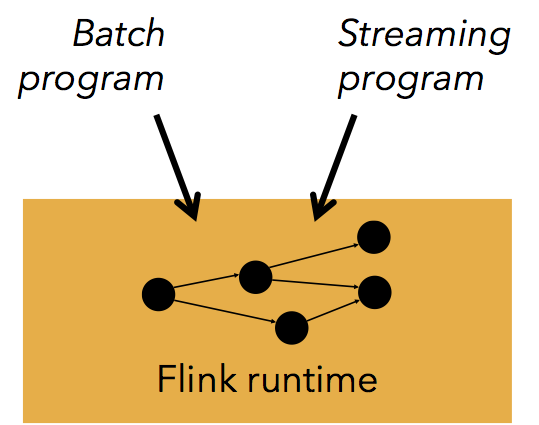
Flink 的容错机制是基于 Chandy-Lamport distributed snapshots 来实现的。

这种机制是非常轻量级的，允许系统拥有高吞吐率的同时还能提供强一致性的保障。



### Batch 和 Streaming 一个系统*流处理和批处理共用一个引擎*

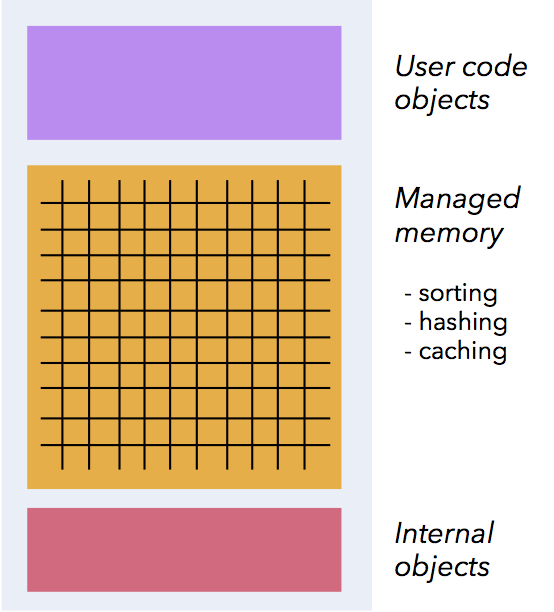
Flink 为流处理和批处理应用公用一个通用的引擎。批处理应用可以以一种特殊的流处理应用高效地运行。



### 内存管理

Flink 在 JVM 中实现了自己的内存管理。

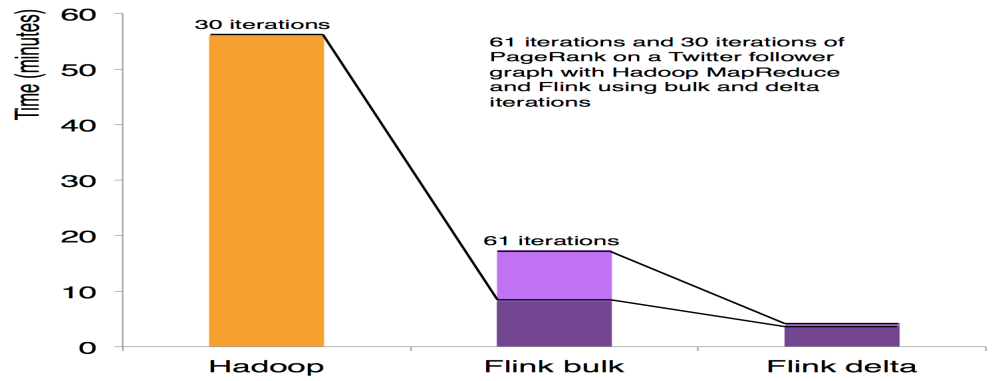
应用可以超出主内存的大小限制，并且承受更少的垃圾收集的开销。



### 迭代和增量迭代

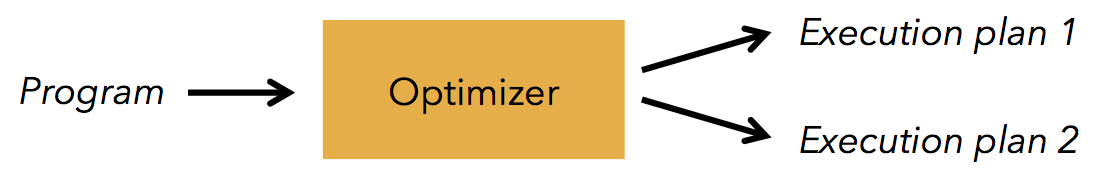
Flink 具有迭代计算的专门支持（比如在机器学习和图计算中）。

增量迭代可以利用依赖计算来更快地收敛。



### 程序调优

批处理程序会自动地优化一些场景，比如避免一些昂贵的操作（如 shuffles 和 sorts），还有缓存一些中间数据。



### API 和 类库

#### 流处理应用

*DataStream* API 支持了数据流上的函数式转换，可以使用自定义的状态和灵活的窗口。

右侧的示例展示了如何以滑动窗口的方式统计文本数据流中单词出现的次数。

WindowWordCount in Flink's DataStream API

**case** **class** **Word(**word**:** **String,** freq**:** **Long)**

**val** texts**:** **DataStream[String]** **=** **...**

**val** counts **=** text

**.**flatMap **{** line **=>** line**.**split**(**"\\W+"**)** **}**

**.**map **{** token **=>** **Word(**token**,** 1**)** **}**

**.**keyBy**(**"word"**)**

**.**timeWindow**(Time.**seconds**(**5**),** **Time.**seconds**(**1**))**

**.**sum**(**"freq"**)**

#### 批处理应用

Flink 的 *DataSet* API 可以使你用 Java 或 Scala 写出漂亮的、类型安全的、可维护的代码。它支持广泛的数据类型，不仅仅是 key/value 对，以及丰富的 operators。

右侧的示例展示了图计算中 PageRank 算法的一个核心循环。

**case** **class** **Page(**pageId**:** **Long,** rank**:** **Double)**

**case** **class** **Adjacency(**id**:** **Long,** neighbors**:** **Array[Long])**

**val** result **=** initialRanks**.**iterate**(**30**)** **{** pages **=>**

pages**.**join**(**adjacency**).**where**(**"pageId"**).**equalTo**(**"pageId"**)** **{**

**(**page**,** adj**,** out **:** **Collector[Page])** **=>** **{**

out**.**collect**(Page(**page**.**id**,** 0.15 **/** numPages**))**

**for** **(**n **<-** adj**.**neighbors**)** **{**

out**.**collect**(Page(**n**,** 0.85**\***page**.**rank**/**adj**.**neighbors**.**length**))**

**}**

**}**

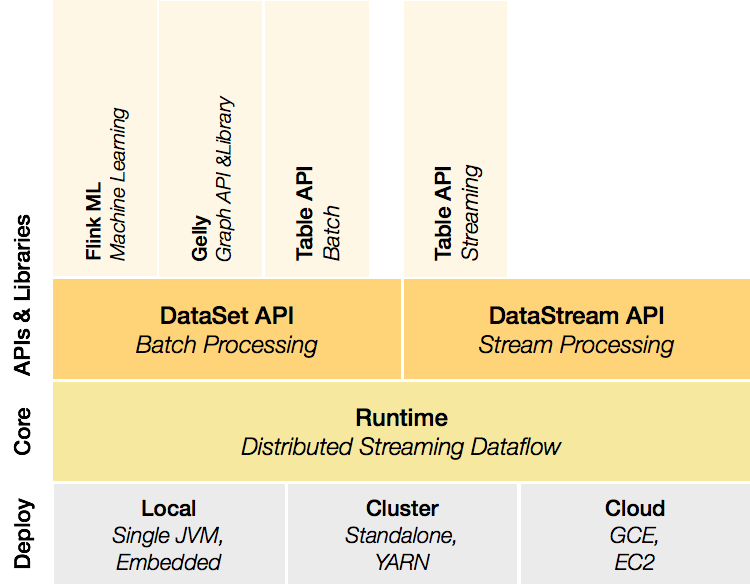
**}**

**.**groupBy**(**"pageId"**).**sum**(**"rank"**)**

**}**

### 类库生态

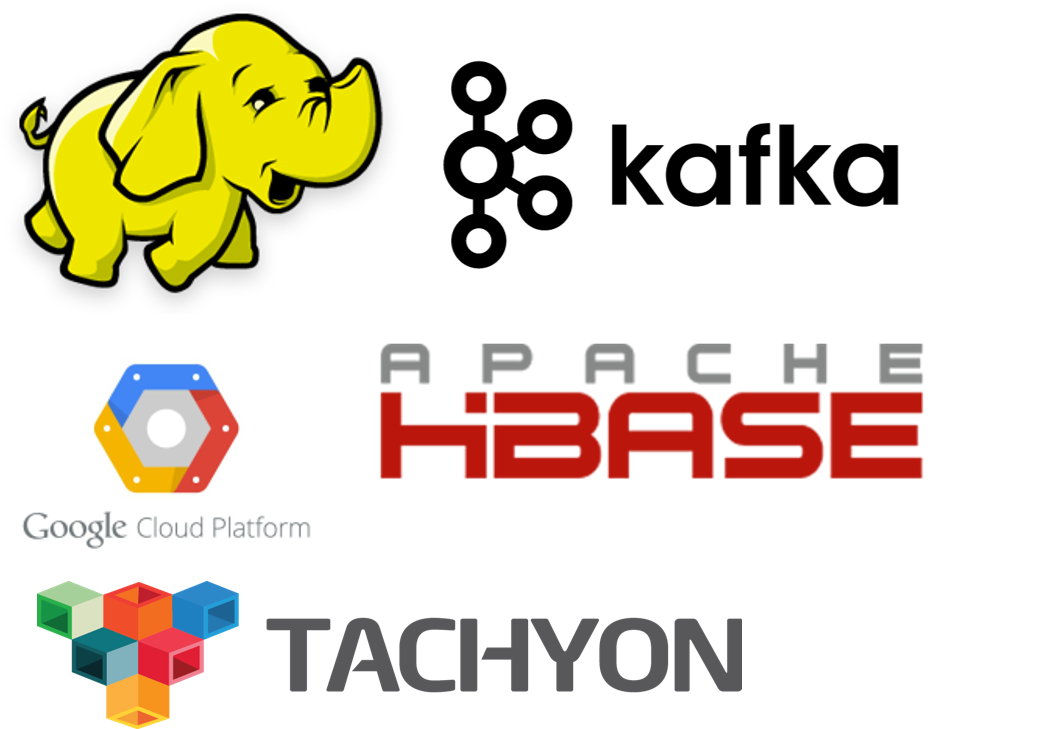
Flink 栈中提供了提供了很多具有高级 API 和满足不同场景的类库：机器学习、图分析、关系式数据处理。当前类库还在 *beta* 状态，并且在大力发展。



### 广泛集成

Flink 与开源大数据处理生态系统中的许多项目都有集成。

Flink 可以运行在 YARN 上，与 HDFS 协同工作，从 Kafka 中读取流数据，可以执行 Hadoop 程序代码，可以连接多种数据存储系统。

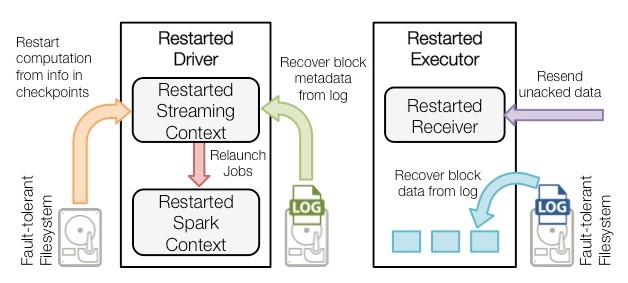


### 部署性

Flink可以单独脱离Hadoop进行部署，部署只依赖Java环境，相对简单。

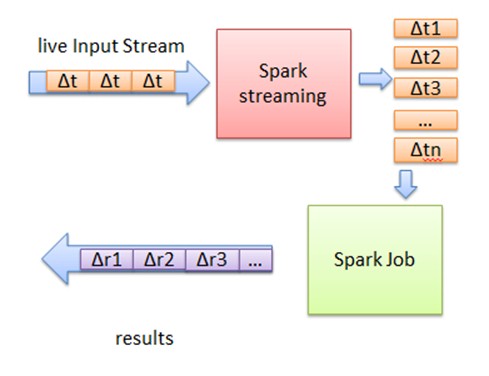
# 2.Spark Streaming架构及特性分析

## 2.1 基本架构



基于是spark core的spark streaming架构。

Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经 过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。



简而言之，Spark Streaming把实时输入数据流以时间片Δt （如1秒）为单位切分成块，Spark Streaming会把每块数据作为一个RDD，并使用RDD操作处理每一小块数据。每个块都会生成一个Spark Job处理，然后分批次提交job到集群中去运行，运行每个job的过程和真正的spark 任务没有任何区别。

### JobScheduler

负责job的调度

JobScheduler是SparkStreaming 所有Job调度的中心， JobScheduler的启动会导致ReceiverTracker和JobGenerator的启动。ReceiverTracker的启动导致运行在Executor端的Receiver启动并且接收数据，ReceiverTracker会记录Receiver接收到的数据meta信息。JobGenerator的启动导致每隔BatchDuration，就调用DStreamGraph生成RDD Graph，并生成Job。JobScheduler中的线程池来提交封装的JobSet对象(时间值，Job，数据源的meta)。Job中封装了业务逻辑，导致最后一个RDD的action被触发，被DAGScheduler真正调度在Spark集群上执行该Job。

### JobGenerator

负责Job的生成

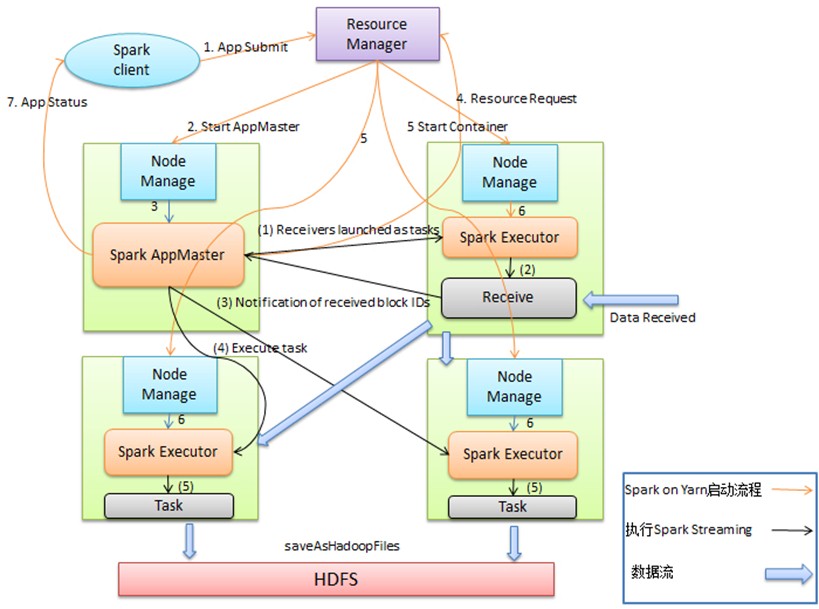
通过定时器每隔一段时间根据Dstream的依赖关系生一个一个DAG图。

### ReceiverTracker

负责数据的接收，管理和分配

ReceiverTracker在启动Receiver的时候他有ReceiverSupervisor,其实现是ReceiverSupervisorImpl, ReceiverSupervisor本身启动的时候会启动Receiver，Receiver不断的接收数据，通过BlockGenerator将数据转换成Block。定时器会不断的把Block数据通过BlockManager或者WAL进行存储，数据存储之后ReceiverSupervisorImpl会把存储后的数据的元数据Metadate汇报给ReceiverTracker,其实是汇报给ReceiverTracker中的RPC实体ReceiverTrackerEndpoint，主要。

## 2.2 基于Yarn层面的架构分析



上图为spark on yarn 的cluster模式，Spark on Yarn启动后，由Spark AppMaster中的driver（在AM的里面会启动driver，主要是StreamingContext对象）把Receiver作为一个Task提交给某一个Spark Executor；Receive启动后输入数据，生成数据块，然后通知Spark AppMaster；Spark AppMaster会根据数据块生成相应的Job，并把Job的Task提交给空闲Spark Executor 执行。图中蓝色的粗箭头显示被处理的数据流，输入数据流可以是磁盘、网络和HDFS等，输出可以是HDFS，数据库等。对比Flink和spark streaming的cluster模式可以发现，都是AM里面的组件（Flink是JM,spark streaming是Driver）承载了task的分配和调度，其他container承载了任务的执行（Flink是TM，spark streaming是Executor），不同的是spark streaming每个批次都要与driver进行通信来进行重新调度，这样延迟性远低于Flink。

### **具体实现**

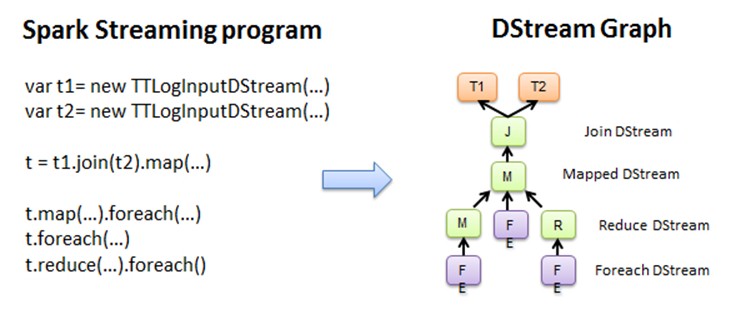


图2.1 Spark Streaming程序转换为DStream Graph

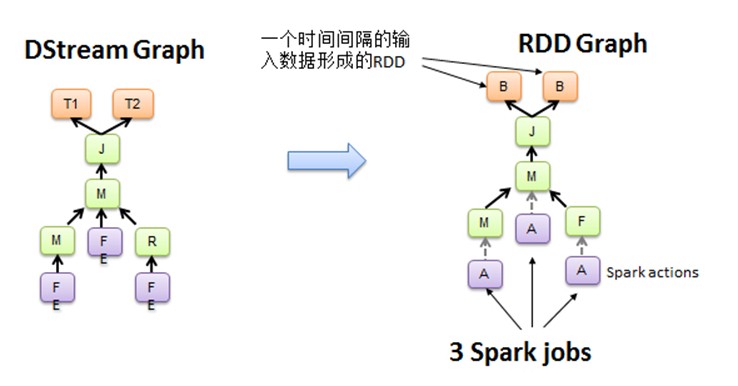


图2.2 DStream Graph转换为RDD的Graph

Spark Core处理的每一步都是基于RDD的，RDD之间有依赖关系。下图中的RDD的DAG显示的是有3个Action，会触发3个job，RDD自下向上依 赖，RDD产生job就会具体的执行。从DSteam Graph中可以看到，DStream的逻辑与RDD基本一致，它就是在RDD的基础上加上了时间的依赖。RDD的DAG又可以叫空间维度，也就是说整个 Spark Streaming多了一个时间维度，也可以成为时空维度，使用Spark Streaming编写的程序与编写Spark程序非常相似，在Spark程序中，主要通过操作RDD（Resilient Distributed Datasets弹性分布式数据集）提供的接口，如map、reduce、filter等，实现数据的批处理。而在Spark Streaming中，则通过操作DStream（表示数据流的RDD序列）提供的接口，这些接口和RDD提供的接口类似。Spark Streaming把程序中对DStream的操作转换为DStream Graph，图2.1中，对于每个时间片，DStream Graph都会产生一个RDD Graph；针对每个输出操作（如print、foreach等），Spark Streaming都会创建一个Spark action；对于每个Spark action，Spark Streaming都会产生一个相应的Spark job，并交给JobScheduler。JobScheduler中维护着一个Jobs队列, Spark job存储在这个队列中，JobScheduler把Spark job提交给Spark Scheduler，Spark Scheduler负责调度Task到相应的Spark Executor上执行，最后形成spark的job。

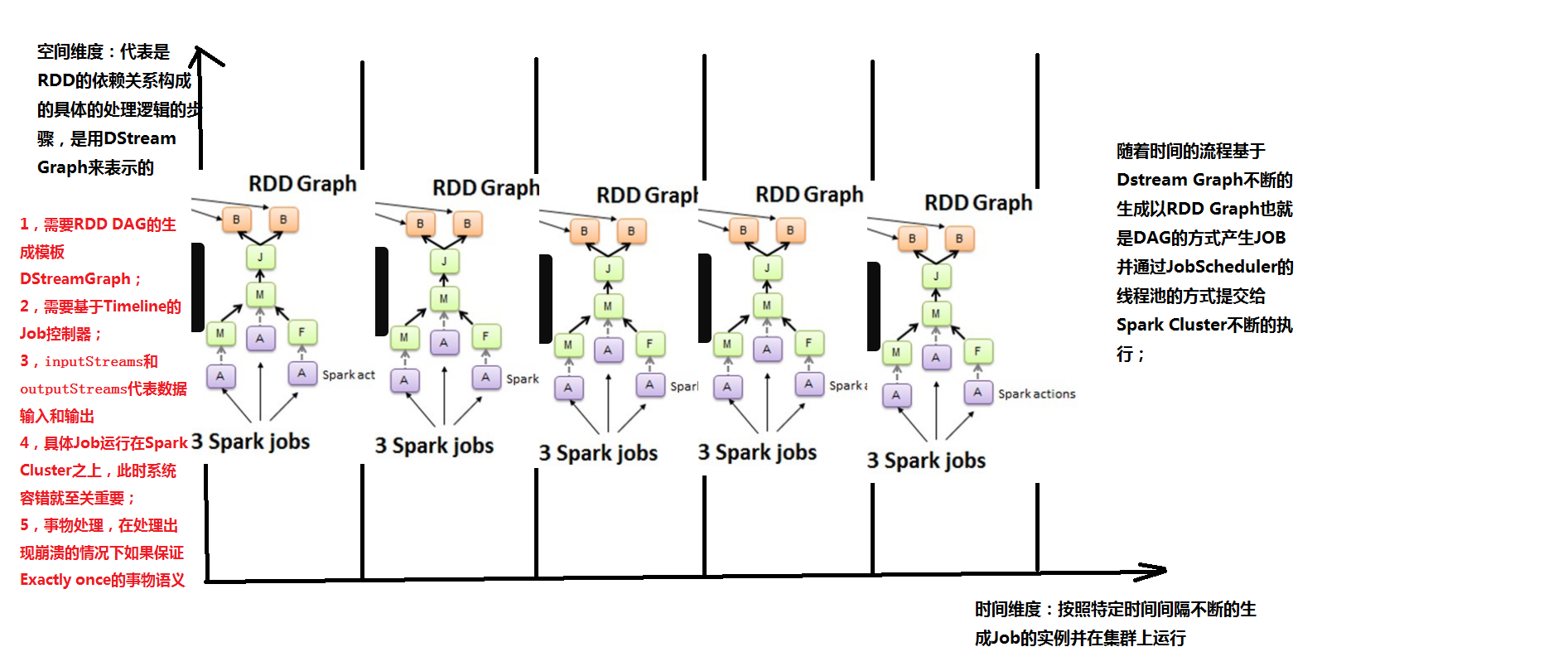
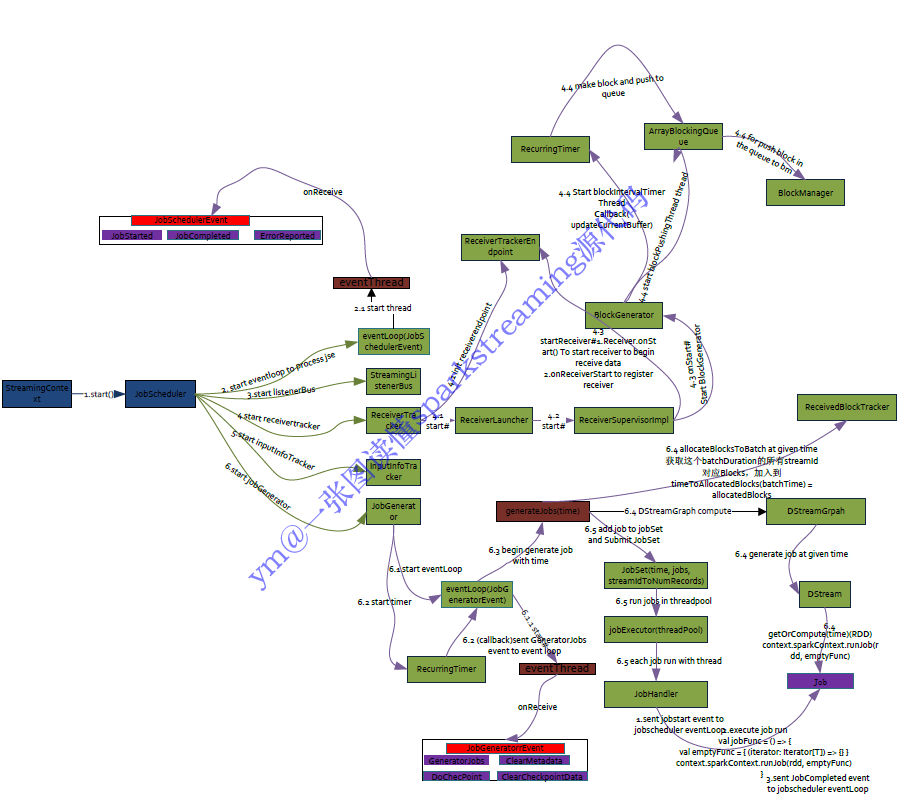


图2.3时间维度生成RDD的DAG

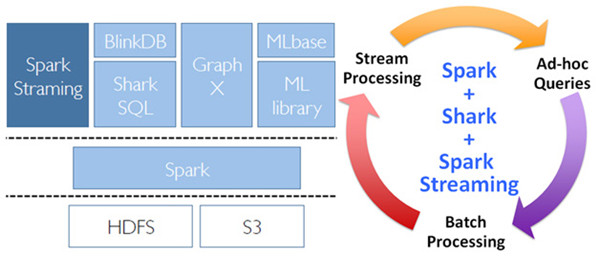
Y轴就是对RDD的操作，RDD的依赖关系构成了整个job的逻辑，而X轴就是时间。随着时间的流逝，固定的时间间隔（Batch Interval）就会生成一个job实例，进而在集群中运行。

### 代码实现



基于spark 1.5的spark streaming源代码解读，基本架构是没怎么变化的。

## 2.3 组件栈

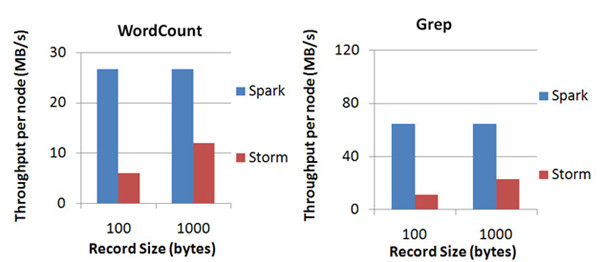


支持从多种数据源获取数据，包括Kafk、Flume、Twitter、ZeroMQ、Kinesis 以及TCP sockets，从数据源获取数据之后，可以使用诸如map、reduce、join和window等高级函数进行复杂算法的处理。最后还可以将处理结果 存储到文件系统，数据库和现场仪表盘。在“One Stack rule them all”的基础上，还可以使用Spark的其他子框架，如集群学习、图计算等，对流数据进行处理。

## 2.4 特性分析

### 吞吐量与延迟性

Spark目前在EC2上已能够线性扩展到100个节点（每个节点4Core），可以以数秒的延迟处理6GB/s的数据量（60M records/s），其吞吐量也比流行的Storm高2～5倍，图4是Berkeley利用WordCount和Grep两个用例所做的测试，在 Grep这个测试中，Spark Streaming中的每个节点的吞吐量是670k records/s，而Storm是115k records/s。



Spark Streaming将流式计算分解成多个Spark Job，对于每一段数据的处理都会经过Spark DAG图分解，以及Spark的任务集的调度过程，其最小的Batch Size的选取在0.5~2秒钟之间（Storm目前最小的延迟是100ms左右），所以Spark Streaming能够满足除对实时性要求非常高（如高频实时交易）之外的所有流式准实时计算场景。

### exactly-once 语义

更加稳定的exactly-once语义支持。

### 反压能力的支持

Spark Streaming 从v1.5开始引入反压机制（back-pressure）,通过动态控制数据接收速率来适配集群数据处理能力.

#### Sparkstreaming如何反压？

简单来说，反压机制需要调节系统接受数据速率或处理数据速率，然而系统处理数据的速率是没法简单的调节。因此，**只能估计当前系统处理数据的速率，调节系统接受数据的速率来与之相匹配。**

#### Flink如何反压？

严格来说，Flink无需进行反压，因为**系统接收数据的速率和处理数据的速率是自然匹配的**。系统接收数据的前提是接收数据的Task必须有空闲可用的Buffer，该数据被继续处理的前提是下游Task也有空闲可用的Buffer。因此，不存在系统接受了过多的数据，导致超过了系统处理的能力。

由此看出，**Spark的micro-batch模型导致了它需要单独引入反压机制**。

#### 反压与高负载

反压通常产生于这样的场景：短时负载高峰导致系统接收数据的速率远高于它处理数据的速率。

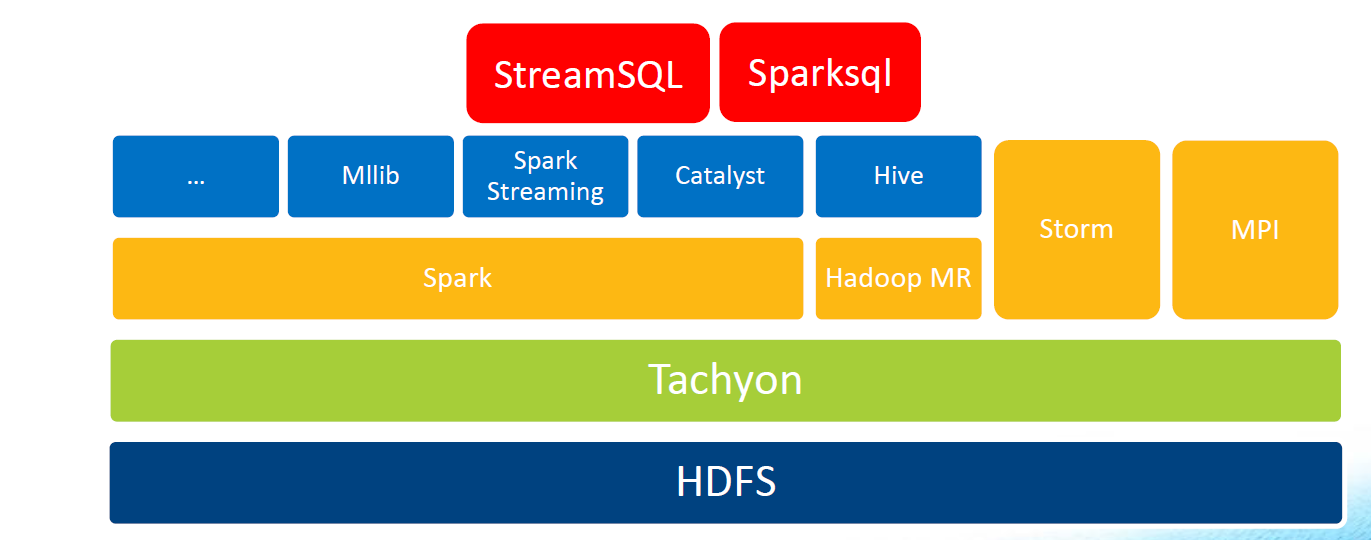
但是，系统能够承受多高的负载是系统数据处理能力决定的，反压机制并不是提高系统处理数据的能力，而是系统所面临负载高于承受能力时如何调节系统接收数据的速率。

### 容错

Driver和executor采用预写日志（WAL）方式去保存状态，同时结合RDD本身的血统的容错机制。

### API 和 类库

Spark 2.0中引入了结构化数据流，统一了SQL和Streaming的API，采用DataFrame作为统一入口，能够像编写普通Batch程序或者直接像操作SQL一样操作Streaming，易于编程。



### 广泛集成

除了可以读取[HDFS](http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html), [Flume](http://flume.apache.org/), [Kafka](http://kafka.apache.org/), [Twitter](https://dev.twitter.com/) and[ZeroMQ](http://zeromq.org/)数据源以外，我们自己也可以定义数据源，支持运行在Yarn，Standalone及EC2上，能够通过Zookeeper，HDFS保证高可用性，处理结果可以直接写到HDFS

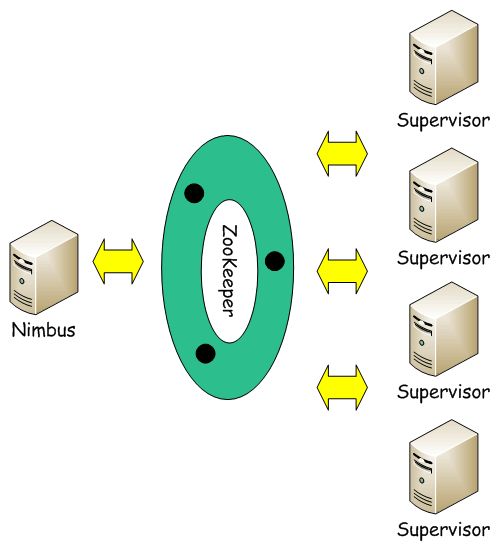
### 部署性

依赖java环境，只要应用能够加载到spark相关的jar包即可。

# 3.Storm架构及特性分析

## 3.1 基本架构

Storm集群采用主从架构方式，主节点是Nimbus，从节点是Supervisor，有关调度相关的信息存储到ZooKeeper集群中。架构如下：



### Nimbus

Storm集群的Master节点，负责分发用户代码，指派给具体的Supervisor节点上的Worker节点，去运行Topology对应的组件（Spout/Bolt）的Task。

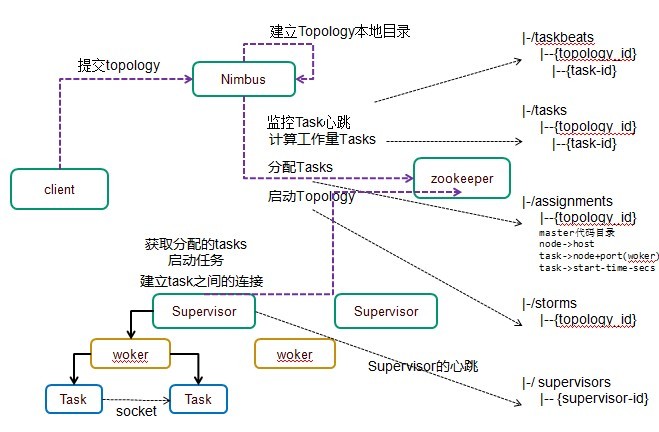
### Supervisor

Storm集群的从节点，负责管理运行在Supervisor节点上的每一个Worker进程的启动和终止。通过Storm的配置文件中的supervisor.slots.ports配置项，可以指定在一个Supervisor上最大允许多少个Slot，每个Slot通过端口号来唯一标识，一个端口号对应一个Worker进程（如果该Worker进程被启动）。

### ZooKeeper

用来协调Nimbus和Supervisor，如果Supervisor因故障出现问题而无法运行Topology，Nimbus会第一时间感知到，并重新分配Topology到其它可用的Supervisor上运行。

### 运行架构



运行流程

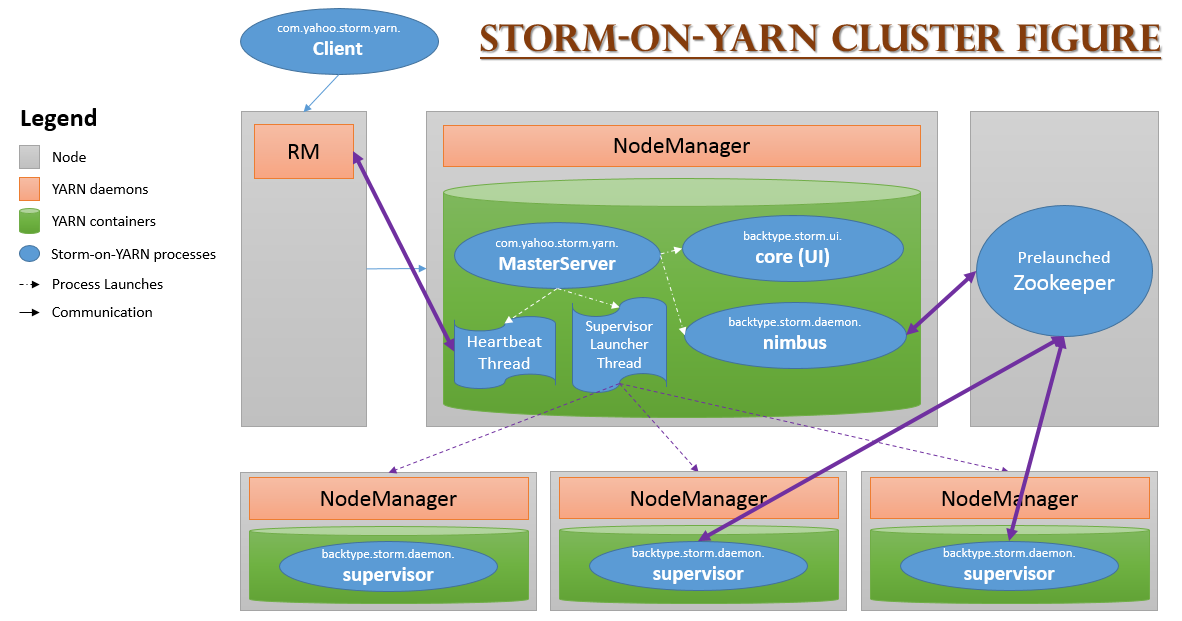
1）户端提交拓扑到nimbus。

2） Nimbus针对该拓扑建立本地的目录根据topology的配置计算task，分配task，在zookeeper上建立assignments节点存储task和supervisor机器节点中woker的对应关系；

在zookeeper上创建taskbeats节点来监控task的心跳；启动topology。

3） Supervisor去zookeeper上获取分配的tasks，启动多个woker进行，每个woker生成task，一个task一个线程；根据topology信息初始化建立task之间的连接;Task和Task之间是通过zeroMQ管理的；后整个拓扑运行起来。

## 3.2 基于Yarn层面的架构



在YARN上开发一个应用程序，通常只需要开发两个组件，分别是客户端和ApplicationMaster，其中客户端主要作用是提交应用程序到YARN上，并和YARN和ApplicationMaster进行交互，完成用户发送的一些指令；而ApplicationMaster则负责向YARN申请资源，并与NodeManager通信，启动任务。

不修改任何Storm源代码即可将其运行在YARN之上，最简单的实现方法是将Storm的各个服务组件（包括Nimbus和Supervisor）作为单独的任务运行在YARN上，而Zookeeper作为一个公共的服务运行在YARN集群之外的几个节点上。

1）通过YARN-Storm Client将Storm Application提交到YARN的RM上；

2）RM为YARN-Storm ApplicationMaster申请资源，并将其运行在一个节点上（Nimbus）；

3）YARN-Storm ApplicationMaster 在自己内部启动Nimbus和UI服务；

4)YARN-Storm ApplicationMaster 根据用户配置向RM申请资源，并在申请到的Container中启动Supervisor服务；

5)与向普通Storm集群提交Topology一样，用户直接与运行在Nimbus交互，提交Topology。

## 3.3组件栈

## 3.4 特性分析

### 简单的编程模型。

类似于MapReduce降低了并行批处理复杂性，Storm降低了进行实时处理的复杂性。

### 服务化

一个服务框架,支持热部署,即时上线或下线App.

### 可以使用各种编程语言

你可以在Storm之上使用各种编程语言。默认支持Clojure、Java、Ruby和Python。要增加对其他语言的支持，只需实现一个简单的Storm通信协议即可。

### 容错性

Storm会管理工作进程和节点的故障。

### 水平扩展

计算是在多个线程、进程和服务器之间并行进行的。

### 可靠的消息处理

Storm保证每个消息至少能得到一次完整处理。任务失败时，它会负责从消息源重试消息。

### 快速

系统的设计保证了消息能得到快速的处理，使用ZeroMQ作为其底层消息队列。

### 本地模式

Storm有一个“本地模式”，可以在处理过程中完全模拟Storm集群。这让你可以快速进行开发和单元测试。

### 部署性

依赖于Zookeeper进行任务状态的维护，必须首先部署Zookeeper。

## 4．三种框架的对比分析

|  |  |  |  |
| --- | --- | --- | --- |
| Apache | Flink | SparkStreaming | Storm |
| 架构 | 架构介于spark和storm之间，主从结构与spark streaming相似，DataFlow Grpah与Storm相似，数据流可以被表示为一个有向图。 每个顶点是一个用户定义的运算，每向边表示数据的流动。  Native | 架构依赖spark，主从模式，每个Batch处理都依赖主（driver），可以理解为时间维度上的spark DAG。  Micro-Batch | 主从模式，且依赖ZK，处理过程中对主的依赖不大。  Native |
| 容错 | 基于Chandy-Lamport distributed snapshots checkpoint机制  Medium | WAL及RDD 血统机制  High | Records ACK  Medium |
| 处理模型与延迟 | 单条事件处理  亚秒级低延迟 | 一个事件窗口内的所有事件。  秒级高延迟 | 每次传入的一个事件  亚秒级低延迟 |
| 吞吐量 | High | High | Low |
| 数据处理保证 | exactly once  High | exactly once（实现采用Chandy-Lamport 算法，即marker-checkpoint ）  High | at least once(实现采用record-level acknowledgments)，Trident可以支持storm 提供exactly once语义。  Medium |
| 高级API | Flink 栈中提供了提供了很多具有高级 API 和满足不同场景的类库：机器学习、图分析、关系式数据处理  High | 能够很容易的对接Spark生态栈里面的组件，同时能够对接主流的消息传输组件及存储系统。  High | 应用需要按照特定的storm定义的规则编写。  Low |
| 易用性 | 支持SQL Steaming，Batch和STREAMING采用统一编程框架  High | 支持SQL Steaming  Batch和STREAMING采用统一编程框架。  High | 不支持SQL Steaming  Low |
| 成熟性 | 新兴项目，处于发展阶段。  Low | 已经发展一段时间  Medium | 相对较早的流系统，比较稳定。  High |
| 社区活跃度 | 212 contributor，活跃度呈上升趋势。  Medium | 937 contirbutor  High | 216 contributors，活跃度比较稳定。  Medium |
| 部署性 | 部署相对简单，只依赖JRE环境  Low | 部署相对简单，只依赖JRE环境  Low | 依赖JRE环境和Zookeeper  High |

### 对比分析

如果对延迟要求不高的情况下，建议使用Spark Streaming，丰富的高级API，使用简单，天然对接Spark生态栈中的其他组件，吞吐量大，部署简单，UI界面也做的更加智能，社区活跃度较高，有问题响应速度也是比较快的，比较适合做流式的ETL，而且Spark的发展势头也是有目共睹的，相信未来性能和功能将会更加完善。

如果对延迟性要求比较高的话，建议可以尝试下Flink，Flink是目前发展比较火的一个流系统，采用原生的流处理系统，保证了低延迟性，在API和容错上也是做的比较完善，使用起来相对来说也是比较简单的，部署容易，而且发展势头也越来越好，相信后面社区问题的响应速度应该也是比较快的。

个人对Flink是比较看好的，因为原生的流处理理念，在保证了低延迟的前提下，性能还是比较好的，且越来越易用，社区也在不断发展。