# 一、Storm

## （一）什么是Storm?

Storm为分布式实时计算提供了一组通用原语，可被用于“流处理”之中，实时处理消息并更新数据库。这是管理队列及工作者集群的另一种方式。 Storm也可被用于“连续计算”（continuous computation），对数据流做连续查询，在计算时就将结果以流的形式输出给用户。它还可被用于“分布式RPC”，以并行的方式运行昂贵的运算。

Storm可以方便地在一个计算机集群中编写与扩展复杂的实时计算，Storm用于实时处理，就好比 Hadoop 用于批处理。Storm保证每个消息都会得到处理，而且它很快——在一个小集群中，每秒可以处理数以百万计的消息。更棒的是你可以使用任意编程语言来做开发。

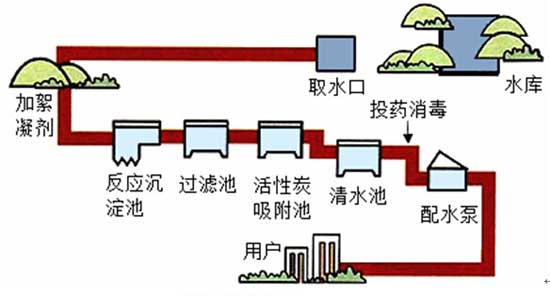
## （二）离线计算和流式计算

1. **离线计算**

* 离线计算：批量获取数据、批量传输数据、周期性批量计算数据、数据展示
* 代表技术：Sqoop批量导入数据、HDFS批量存储数据、MapReduce批量计算、Hive

1. **流式计算**

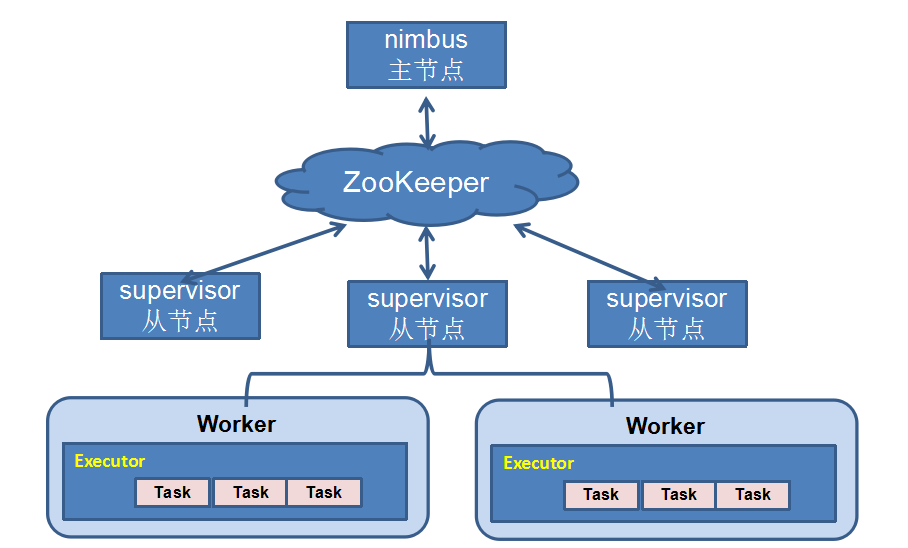
* 流式计算：数据实时产生、数据实时传输、数据实时计算、实时展示
* 代表技术：Flume实时获取数据、Kafka/metaq实时数据存储、Storm/JStorm实时数据计算、Redis实时结果缓存、持久化存储(mysql)。
* 一句话总结：将源源不断产生的数据实时收集并实时计算，尽可能快的得到计算结果

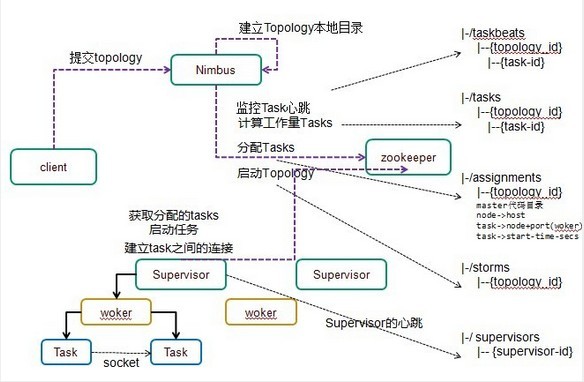


1. **Storm与Hadoop的区别**

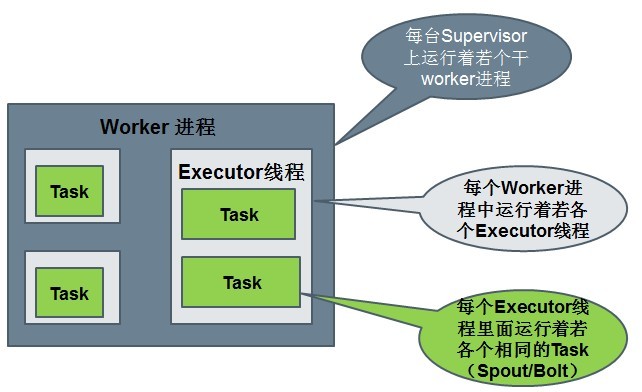
|  |  |
| --- | --- |
| Storm用于实时计算 | Hadoop用于离线计算 |
| Storm处理的数据保存在内存中，源源不断 | Hadoop处理的数据保存在文件系统中，一批一批 |
| Storm的数据通过网络传输进来 | Hadoop的数据保存在磁盘中 |
| Storm与Hadoop的编程模型相似 | |

## （三）Storm的体系结构

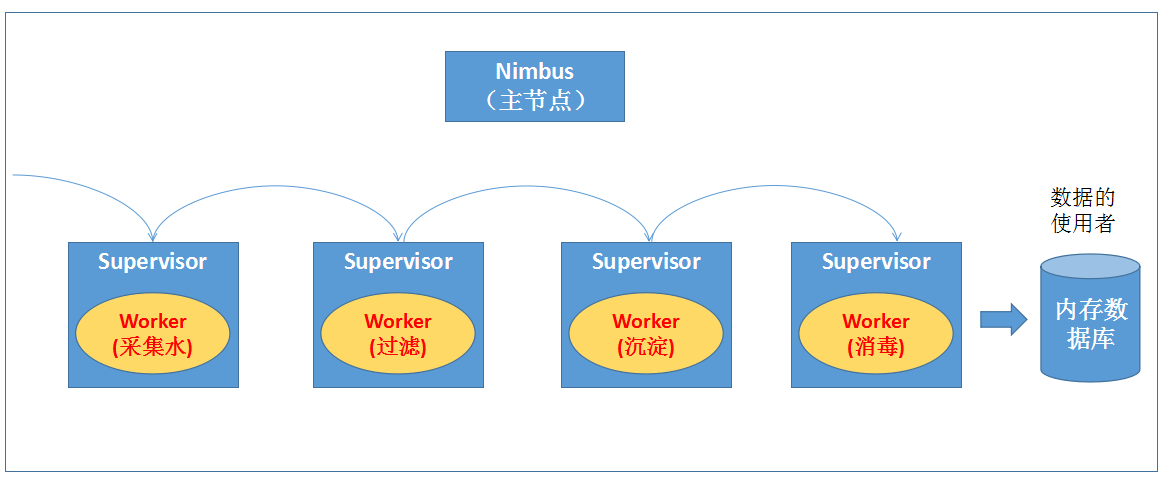




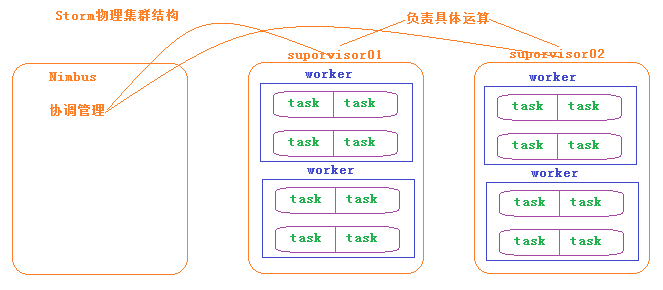
* **Nimbus**：负责资源分配和任务调度。
* **Supervisor**：负责接受nimbus分配的任务，启动和停止属于自己管理的worker进程。通过配置文件设置当前supervisor上启动多少个worker。
* **Worker**：运行具体处理组件逻辑的进程。Worker运行的任务类型只有两种，一种是Spout任务，一种是Bolt任务。
* **Executor**：Storm 0.8之后，Executor为Worker进程中的具体的物理线程，同一个Spout/Bolt的Task可能会共享一个物理线程，一个Executor中只能运行隶属于同一个Spout/Bolt的Task。
* **Task**：worker中每一个spout/bolt的线程称为一个task. 在storm0.8之后，task不再与物理线程对应，不同spout/bolt的task可能会共享一个物理线程，该线程称为executor。



## （四）Storm的运行机制

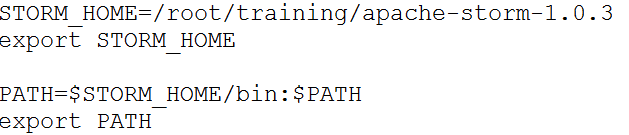


* + 整个处理流程的组织协调不用用户去关心，用户只需要去定义每一个步骤中的具体业务处理逻辑
  + 具体执行任务的角色是Worker，Worker执行任务时具体的行为则有我们定义的业务逻辑决定

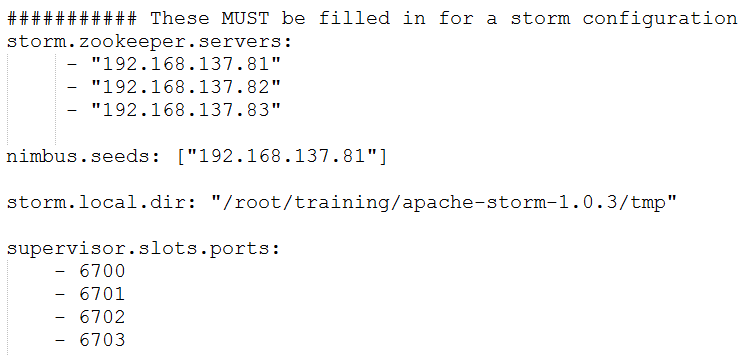


## （五）Storm的安装配置

* 解压：tar -zxvf apache-storm-1.0.3.tar.gz -C ~/training/
* 设置环境变量



* 编辑配置文件：$STORM\_HOME/conf/storm.yaml



* + 注意：如果要搭建Storm的HA，只需要在nimbus.seeds中设置多个nimbus即可。
* 把安装包复制到其他节点上。

## （六）启动和查看Storm

* 在nimbus.host所属的机器上启动 nimbus服务和logviewer服务
  + storm nimbus &
  + storm logviewer &
* 在nimbus.host所属的机器上启动ui服务
  + storm ui &
* 在其它个点击上启动supervisor服务和logviewer服务
  + storm supervisor &
  + storm logviewer &
* 查看storm集群：访问nimbus.host:/8080，即可看到storm的ui界面



## （七）Storm的常用命令

有许多简单且有用的命令可以用来管理拓扑，它们可以提交、杀死、禁用、再平衡拓扑。

1. 提交任务命令格式：storm jar 【jar路径】 【拓扑包名.拓扑类名】 【拓扑名称】



1. 杀死任务命令格式：storm kill 【拓扑名称】 -w 10

（执行kill命令时可以通过-w [等待秒数]指定拓扑停用以后的等待时间）

**storm kill topology-name -w 10**

1. 停用任务命令格式：storm deactivte 【拓扑名称】

**storm deactivte topology-name**

1. 启用任务命令格式：storm activate【拓扑名称】

**storm activate topology-name**

1. 重新部署任务命令格式：storm rebalance 【拓扑名称】

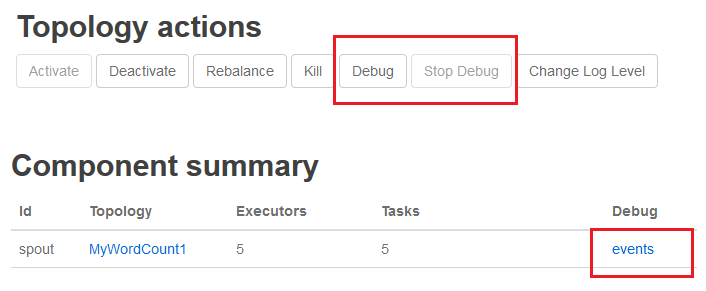
**storm rebalance topology-name**

再平衡使你重分配集群任务。这是个很强大的命令。比如，你向一个运行中的集群增加了节点。再平衡命令将会停用拓扑，然后在相应超时时间之后重分配工人，并重启拓扑。

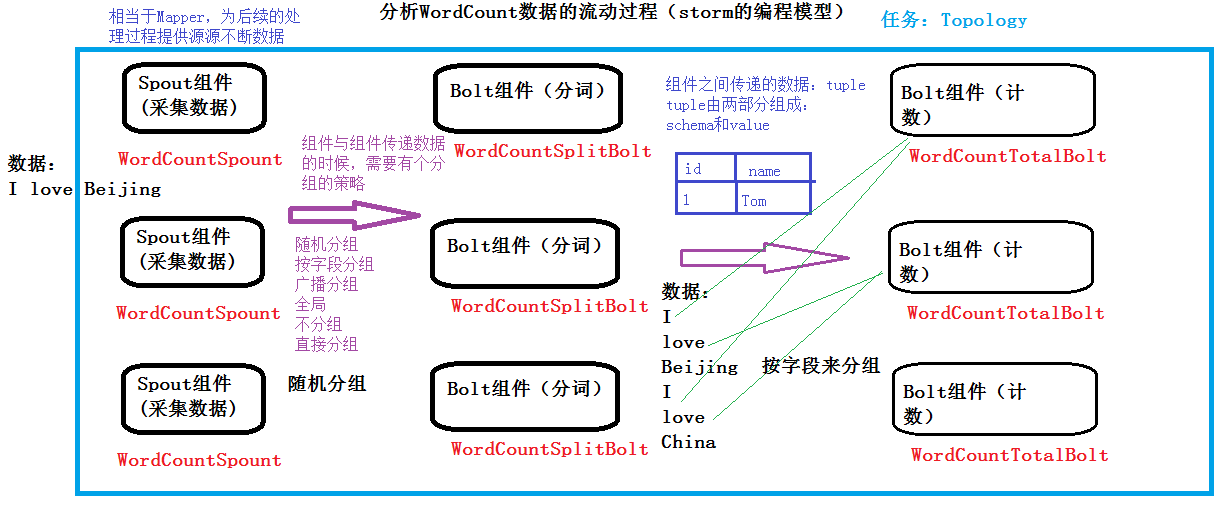
## （八）Demo演示：WordCount及流程分析

通过查看Storm UI上每个组件的events链接，可以查看Storm的每个组件（spout、blot）发送的消息。但Storm的event logger的功能默认是禁用的，需要在配置文件中设置：topology.eventlogger.executors: 1，具体说明如下：

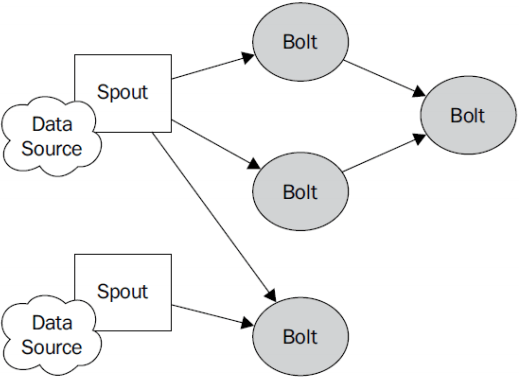
* "topology.eventlogger.executors": 0 默认，禁用
* "topology.eventlogger.executors": 1 一个topology分配一个Event Logger.
* "topology.eventlogger.executors": nil 每个worker.分配一个Event Logger



**WordCount的数据流程分析**



## （九）Storm的编程模型



* **Topology**：Storm中运行的一个实时应用程序的名称。（拓扑）
* **Spout**：在一个topology中获取源数据流的组件。
  + 通常情况下spout会从外部数据源中读取数据，然后转换为topology内部的源数据。
* **Bolt**：接受数据然后执行处理的组件,用户可以在其中执行自己想要的操作。
* **Tuple**：一次消息传递的基本单元，理解为一组消息就是一个Tuple。
* **Stream**：表示数据的流向。
* **StreamGroup**：数据分组策略
  + Shuffle Grouping ：随机分组，尽量均匀分布到下游Bolt中
  + Fields Grouping ：按字段分组，按数据中field值进行分组；相同field值的Tuple被发送到相同的Task
  + All grouping ：广播
  + Global grouping ：全局分组，Tuple被分配到一个Bolt中的一个Task，实现事务性的Topology。
  + None grouping ：不分组
  + Direct grouping ：直接分组 指定分组

## （十）Storm编程案例：WordCount

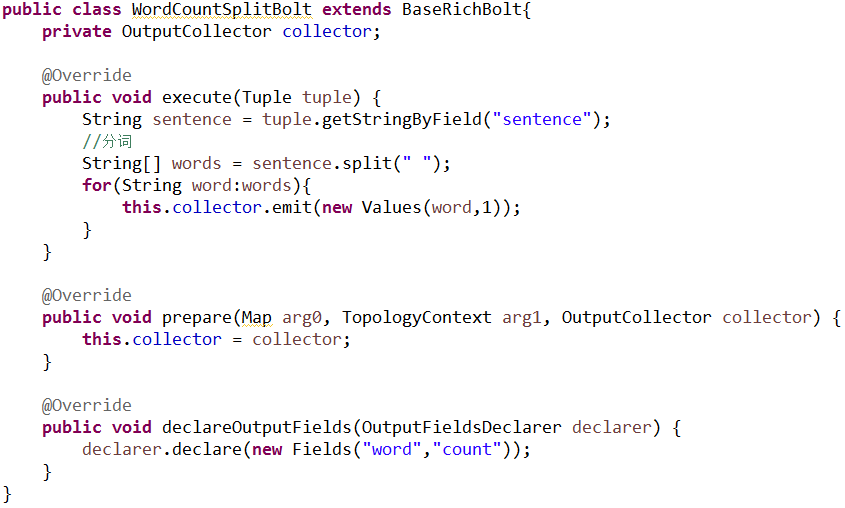
流式计算一般架构图：



* Flume用来获取数据。
* Kafka用来临时保存数据。
* Strom用来计算数据。
* Redis是个内存数据库，用来保存数据。
  + **创建Spout（WordCountSpout）组件采集数据，作为整个Topology的数据源**



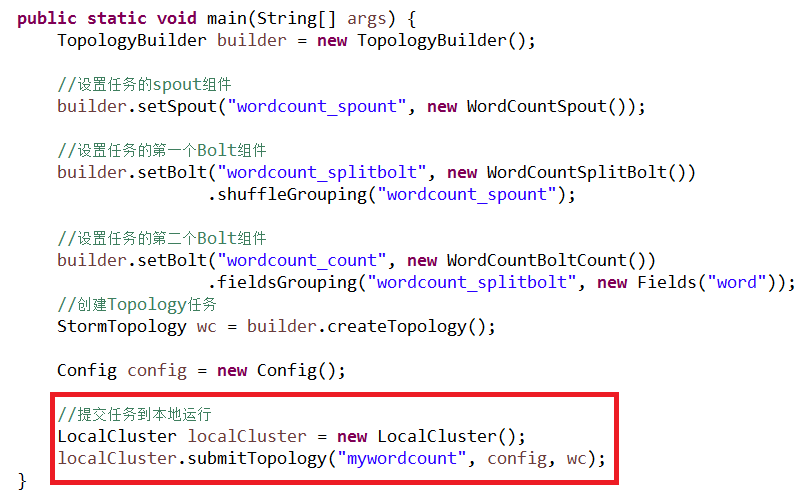
* + **创建Bolt（WordCountSplitBolt）组件进行分词操作**



* + **创建Bolt（WordCountBoltCount）组件进行单词计数作**



* + **创建主程序Topology（WordCountTopology），并提交到本地运行**



* + **也可以将主程序Topology（WordCountTopology）提交到Storm集群运行**



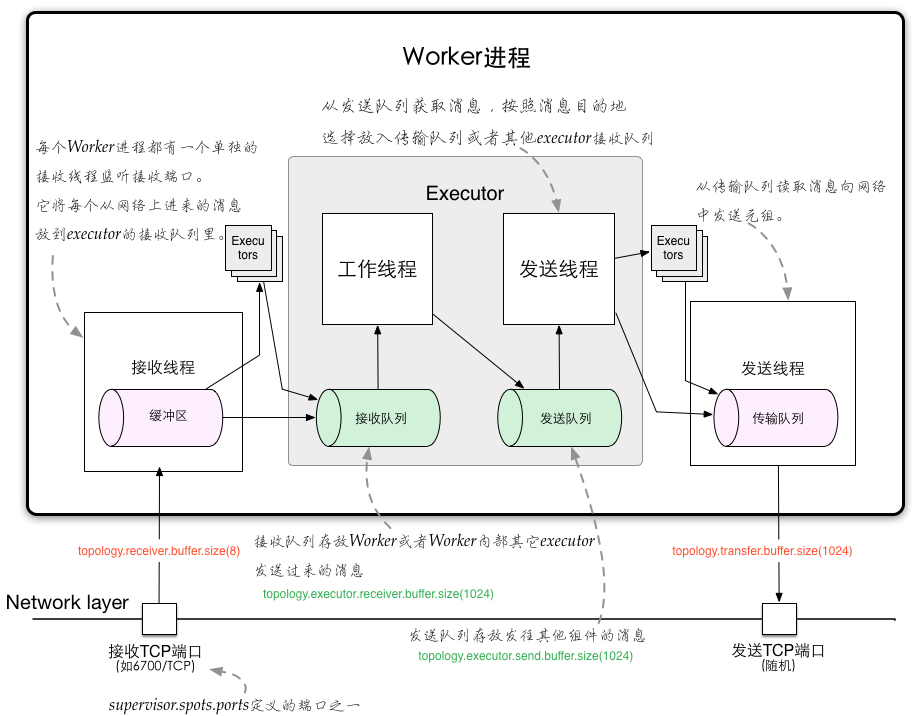
## （十一）Storm集群在ZK上保存的数据结构

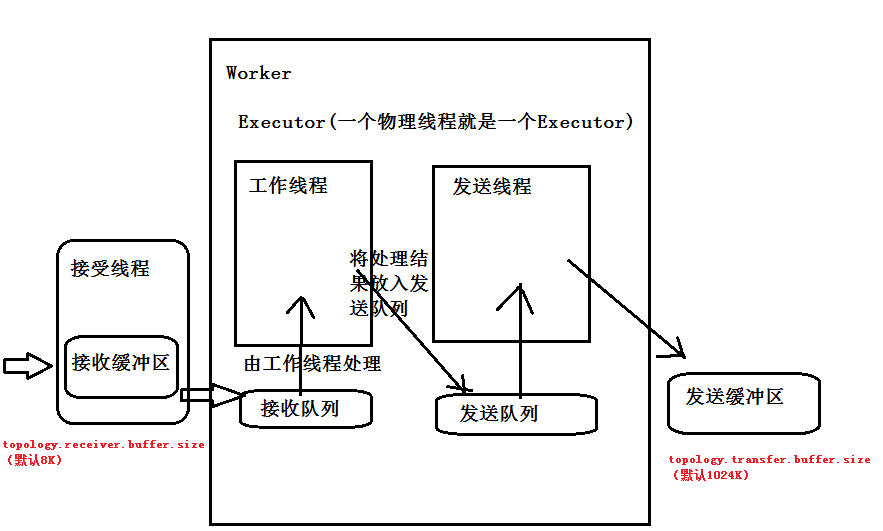


## （十二）Storm集群任务提交流程



## （十三）Storm内部通信机制





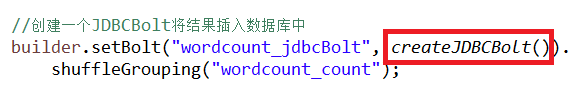
## （十四）集成Storm

### （1）与JDBC集成

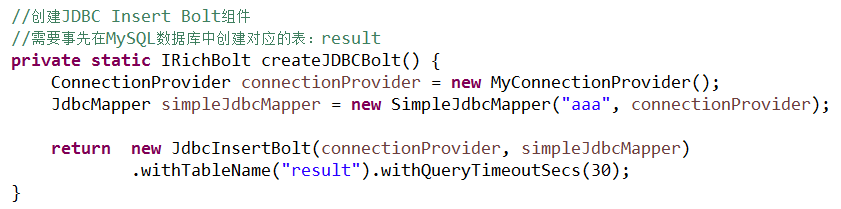
* 将Storm Bolt处理的结果插入MySQL数据库中
* 需要依赖的jar包
  + $STORM\_HOME\external\sql\storm-sql-core\\*.jar
  + $STORM\_HOME\external\storm-jdbc\storm-jdbc-1.0.3.jar
  + mysql的驱动
  + commons-lang3-3.1.jar



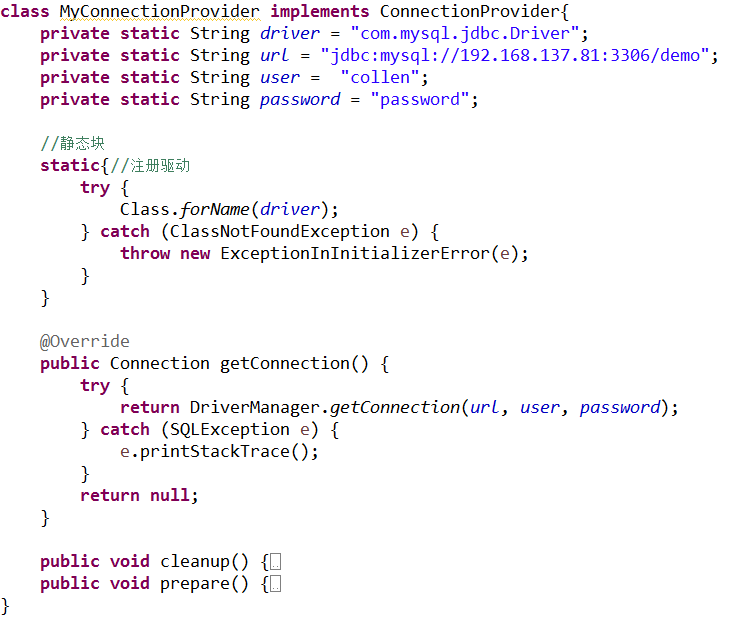
* 与JDBC集成的代码实现
  + 修改主程序WordCountTopology，增加如下代码：



增加一个新方法创建JDBCBolt组件



* + 实现ConnectionProvider接口



* + 修改WordCountBoltCount组件，将统计后的结果发送给下一个组件写入MySQL



### （2）与Redis集成

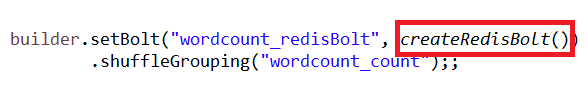
Redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。与Memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

Redis 是一个高性能的key-value数据库。Redis的出现，很大程度补偿了memcached这类key/value存储的不足，在部分场合可以对关系数据库起到很好的补充作用。它提供了Java，C/C++，C#，PHP，JavaScript，Perl，Object-C，Python，Ruby，Erlang等客户端，使用很方便。[1]

Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。



* **修改代码：WordCountTopology.java**





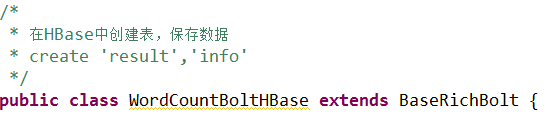
### （3）与HDFS集成

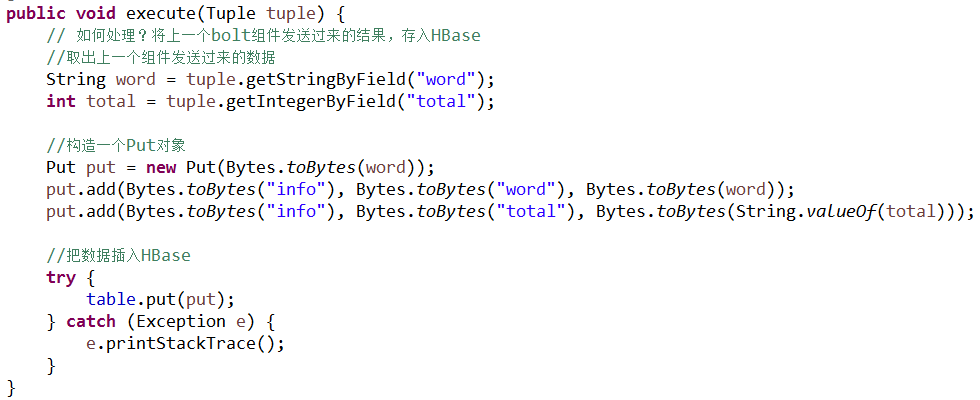
* + - 需要的jar包：
      * $STORM\_HOME\external\storm-hdfs\storm-hdfs-1.0.3.jar
      * HDFS相关的jar包
    - 开发新的bolt组件

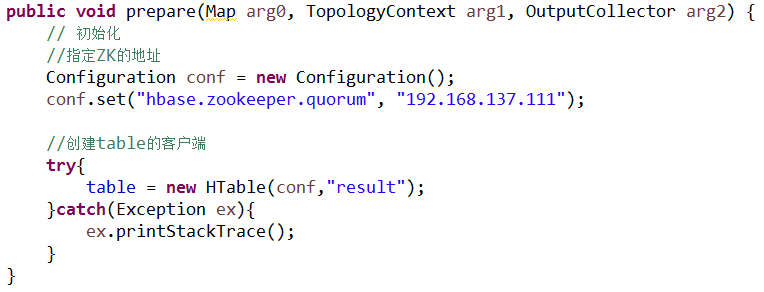


### （4）与HBase集成

* + - 需要的jar包：HBase的相关包
    - 开发新的bolt组件（WordCountBoltHBase.java）







### （5）与Apache Kafka集成

* **注意：需要把slf4j-log4j12-1.6.4.jar包去掉，有冲突（有两个）**



### （6）与Hive集成

* + - **由于集成Storm和Hive依赖的jar较多，并且冲突的jar包很多，强烈建议使用Maven来搭建新的工程。**

<dependencies>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>1.0.3</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-hive</artifactId>

<version>1.0.3</version>

<type>jar</type>

</dependency>

</dependencies>

* + - **需要对Hive做一定的配置（在hive-site.xml文件中）：**

<property>

<name>hive.in.test</name>

<value>true</value>

</property>

* + - **需要使用下面的语句在hive中创建表：**

create table wordcount

(word string,total int)

clustered by (word) into 10 buckets

stored as orc TBLPROPERTIES('transactional'='true');

* + - **启动metastore服务：hive --service metastore**
    - **开发新的bolt组件，用于将前一个bolt处理的结果写入Hive**



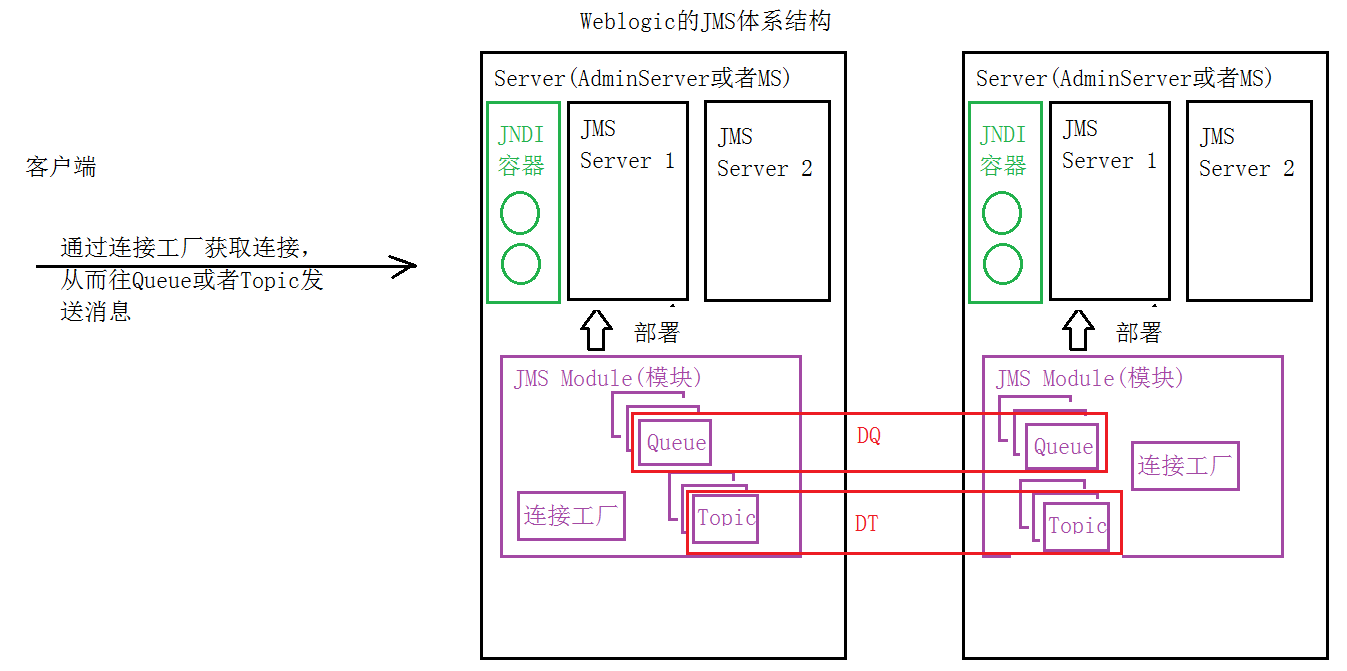
* + - **为了测试的方便，我们依然采用之前随机产生字符串的Spout组件产生数据**

### （7）与JMS集成

JMS即Java消息服务（Java Message Service）应用程序接口，是一个Java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

JMS的两种消息类型：**Queue**和**Topic**

基于Weblogic的JMS架构：







* **需要的weblogic的jar包**



* **permission javax.management.MBeanTrustPermission "register";**

# 二、[Flink](https://www.cnblogs.com/frankdeng/p/9400627.html)

# （一）**简介**

Flink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。

# **统一的批处理与流处理系统**

在大数据处理领域，批处理任务与流处理任务一般被认为是两种不同的任务，一个大数据项目一般会被设计为只能处理其中一种任务，例如Apache Storm、Apache Smaza只支持流处理任务，而Aapche MapReduce、Apache Tez、Apache Spark只支持批处理任务。Spark Streaming是Apache Spark之上支持流处理任务的子系统，看似一个特例，实则不然——Spark Streaming采用了一种micro-batch的架构，即把输入的数据流切分成细粒度的batch，并为每一个batch数据提交一个批处理的Spark任务，所以Spark Streaming本质上还是基于Spark批处理系统对流式数据进行处理，和Apache Storm、Apache Smaza等完全流式的数据处理方式完全不同。通过其灵活的执行引擎，Flink能够同时支持批处理任务与流处理任务。

在执行引擎这一层，流处理系统与批处理系统最大不同在于节点间的数据传输方式。

对于一个流处理系统，其节点间数据传输的标准模型是：当一条数据被处理完成后，序列化到缓存中，然后立刻通过网络传输到下一个节点，由下一个节点继续处理。

而对于一个批处理系统，其节点间数据传输的标准模型是：当一条数据被处理完成后，序列化到缓存中，并不会立刻通过网络传输到下一个节点，当缓存写满，就持久化到本地硬盘上，当所有数据都被处理完成后，才开始将处理后的数据通过网络传输到下一个节点。

这两种数据传输模式是两个极端，对应的是流处理系统对低延迟的要求和批处理系统对高吞吐量的要求。Flink的执行引擎采用了一种十分灵活的方式，同时支持了这两种数据传输模型。Flink以固定的缓存块为单位进行网络数据传输，用户可以通过缓存块超时值指定缓存块的传输时机。如果缓存块的超时值为0，则Flink的数据传输方式类似上文所提到流处理系统的标准模型，此时系统可以获得最低的处理延迟。如果缓存块的超时值为无限大，则Flink的数据传输方式类似上文所提到批处理系统的标准模型，此时系统可以获得最高的吞吐量。

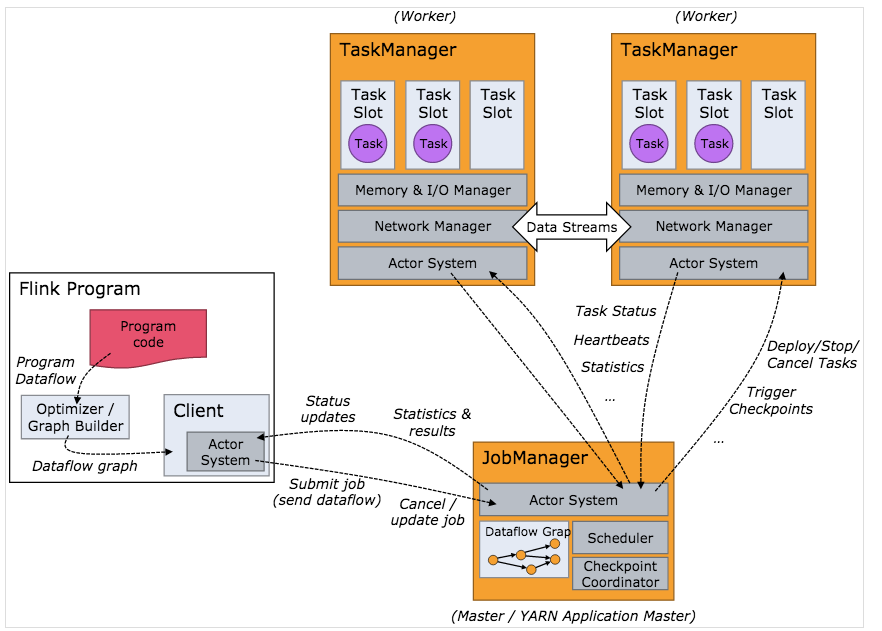
同时缓存块的超时值也可以设置为0到无限大之间的任意值。

缓存块的超时阈值越小，则Flink流处理执行引擎的数据处理延迟越低，但吞吐量也会降低，

反之亦然。通过调整缓存块的超时阈值，用户可根据需求灵活地权衡系统延迟和吞吐量。

# **架构**

要了解一个系统，一般都是从架构开始。我们关心的问题是：系统部署成功后各个节点都启动了哪些服务，各个服务之间又是怎么交互和协调的。下方是 Flink 集群启动后架构图。



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。

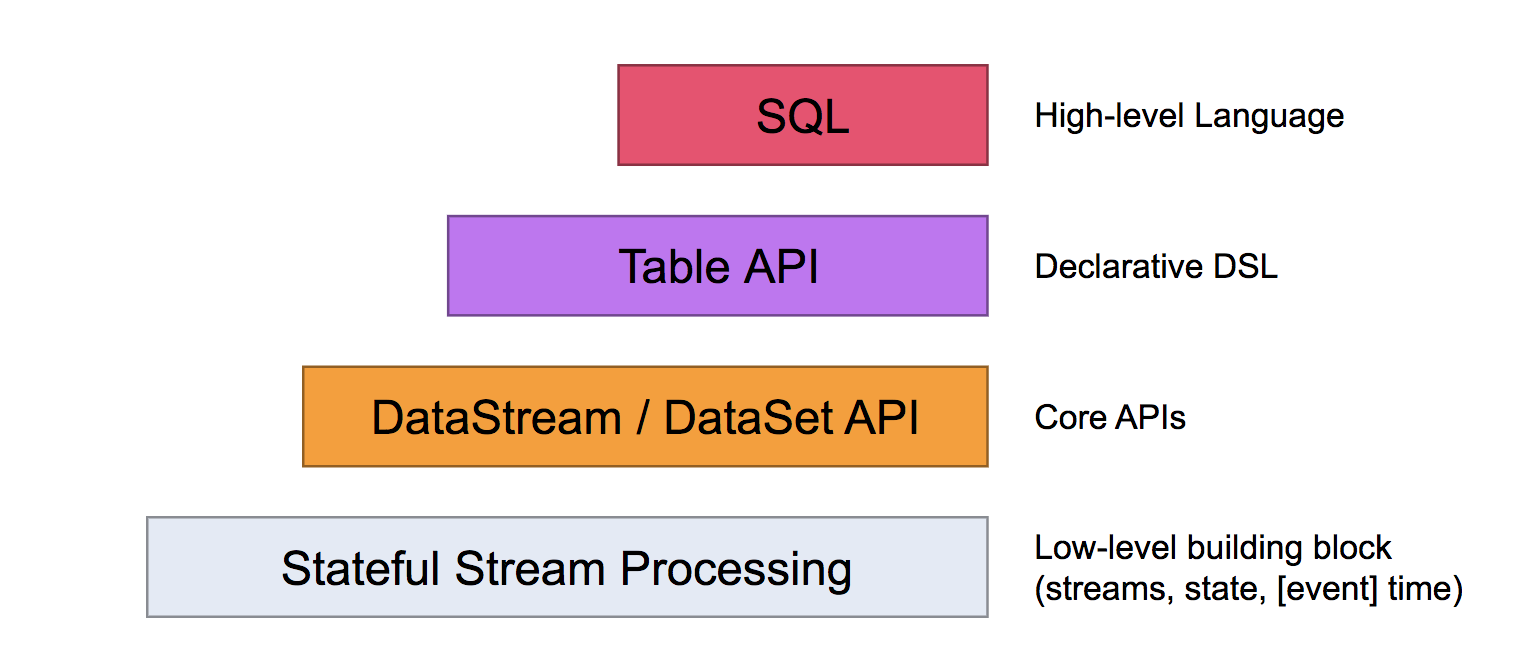
Client 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。

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。虽然这种方式可以有效提高 CPU 利用率，但是个人不太喜欢这种设计，因为不仅缺乏资源隔离机制，同时也不方便调试。类似 Storm 的进程模型，一个JVM 中只跑该 Job 的 Tasks 实际应用中更为合理。

flink编程模型：



# [（四）CentOS7.5搭建Flink1.6.1分布式集群](https://www.cnblogs.com/frankdeng/p/9400627.html)

## （1）Flink的下载

安装包下载地址：[http://flink.apache.org/downloads.html](http://flink.apache.org/downloads.html" \t "_blank)  ，选择对应Hadoop的Flink版本下载

[itstar@bigdata11 software]$ wget http://mirrors.tuna.tsinghua.edu.cn/apache/flink/flink-1.6.1/flink-1.6.1-bin-hadoop28-scala\_2.11.tgz

[itstar@bigdata11 software]$ ll

-rw-rw-r-- 1 itstar itstar 301867081 Sep 15 15:47 flink-1.6.1-bin-hadoop28-scala\_2.11.tgz

**Flink 有三种部署模式，分别是 Local、Standalone Cluster 和 Yarn Cluster。**

## （2）Local模式

对于 Local 模式来说，JobManager 和 TaskManager 会公用一个 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。实际应用中大多使用 Standalone 或者 Yarn Cluster，而local模式只是将安装包解压启动（./bin/start-cluster.sh）即可，在这里不在演示。

## （3）Standalone 模式

**快速入门教程地址：**

**[https://ci.apache.org/projects/flink/flink-docs-release-1.6/quickstart/setup\_quickstart.html](https://ci.apache.org/projects/flink/flink-docs-release-1.6/quickstart/setup_quickstart.html" \t "_blank)**

**1.  软件要求**

* Java 1.8.x或更高版本，
* ssh（必须运行sshd才能使用管理远程组件的Flink脚本）

集群部署规划

|  |  |  |  |
| --- | --- | --- | --- |
| 节点名称 | master | worker | zookeeper |
| bigdata11 | master |  | zookeeper |
| bigdata12 | master | worker | zookeeper |
| bigdata13 |  | worker | zookeeper |

**2. 解压**

[itstar@bigdata11 software]$ tar zxvf flink-1.6.1-bin-hadoop27-scala\_2.11.tgz -C /opt/module/

[itstar@bigdata11 software]$ cd /opt/module/

[itstar@bigdata11 module]$ ll

drwxr-xr-x 8 itstar itstar 125 Sep 15 04:47 flink-1.6.1

**3. 修改配置文件**

[itstar@bigdata11 conf]$ ls

flink-conf.yaml log4j-console.properties log4j-yarn-session.properties logback.xml masters sql-client-defaults.yaml

log4j-cli.properties log4j.properties logback-console.xml logback-yarn.xml slaves zoo.cfg

修改flink/conf/masters，slaves，flink-conf.yaml

[itstar@bigdata11 conf]$ sudo vi masters

bigdata11:8081

[itstar@bigdata11 conf]$ sudo vi slaves

bigdata12

bigdata13

[itstar@bigdata11 conf]$ sudo vi flink-conf.yaml

taskmanager.numberOfTaskSlots：2 //52行 和storm slot类似

jobmanager.rpc.address: bigdata11 //33行

可选配置：

* 每个JobManager（jobmanager.heap.mb）的可用内存量，
* 每个TaskManager（taskmanager.heap.mb）的可用内存量，
* 每台机器的可用CPU数量（taskmanager.numberOfTaskSlots），
* 集群中的CPU总数（parallelism.default）和
* 临时目录（taskmanager.tmp.dirs）

**4. 拷贝安装包到各节点**

[itstar@bigdata11 module]$ scp -r flink-1.6.1/ itstar@bigdata12:`pwd`

[itstar@bigdata11 module]$ scp -r flink-1.6.1/ itstar@bigdata13:`pwd`

**5. 配置环境变量**

配置所有节点Flink的环境变量

[itstar@bigdata11 flink-1.6.1]$ sudo vi /etc/profile

export FLINK\_HOME=/opt/module/flink-1.6.1

export PATH=$PATH:$FLINK\_HOME/bin

[itstar@bigdata11 flink-1.6.1]$ source /etc/profile

**6. 启动flink**

[itstar@bigdata11 flink-1.6.1]$ ./bin/start-cluster.sh

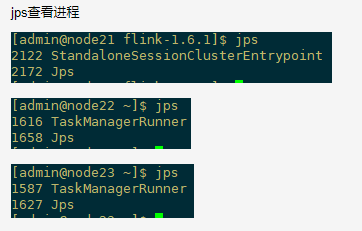
Starting cluster.

Starting standalonesession daemon on host bigdata11.

Starting taskexecutor daemon on host bigdata12.

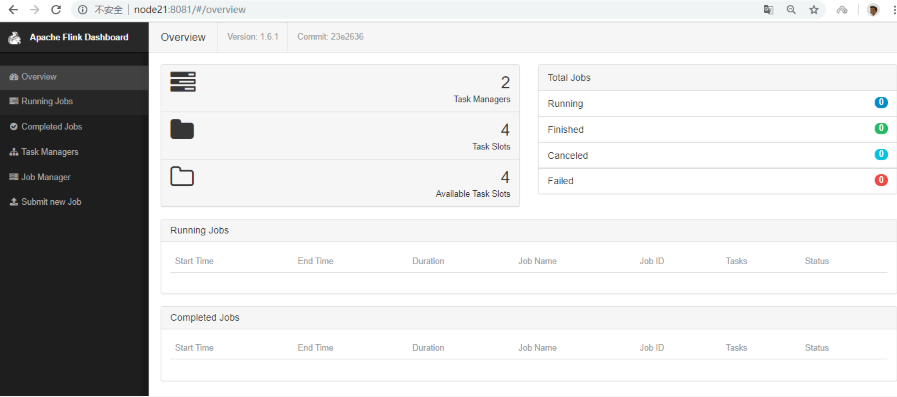
Starting taskexecutor daemon on host bigdata13.

jps查看进程



**7.  WebUI查看**

[http://bigdata11:8081](http://node21:8081/" \t "_blank)



**8. Flink 的 HA**

首先，我们需要知道 Flink 有两种部署的模式，分别是 Standalone 以及 Yarn Cluster 模式。对于 Standalone 来说，Flink 必须依赖于 Zookeeper 来实现 JobManager 的 HA（Zookeeper 已经成为了大部分开源框架 HA 必不可少的模块）。在 Zookeeper 的帮助下，一个 Standalone 的 Flink 集群会同时有多个活着的 JobManager，其中只有一个处于工作状态，其他处于 Standby 状态。当工作中的 JobManager 失去连接后（如宕机或 Crash），Zookeeper 会从 Standby 中选举新的 JobManager 来接管 Flink 集群。

对于 Yarn Cluaster 模式来说，Flink 就要依靠 Yarn 本身来对 JobManager 做 HA 了。其实这里完全是 Yarn 的机制。对于 Yarn Cluster 模式来说，JobManager 和 TaskManager 都是被 Yarn 启动在 Yarn 的 Container 中。此时的 JobManager，其实应该称之为 Flink Application Master。也就说它的故障恢复，就完全依靠着 Yarn 中的 ResourceManager（和 MapReduce 的 AppMaster 一样）。由于完全依赖了 Yarn，因此不同版本的 Yarn 可能会有细微的差异。这里不再做深究。

**1） 修改配置文件**

修改flink-conf.yaml，HA模式下，jobmanager不需要指定，在master file中配置，由zookeeper选出leader与standby。

|  |
| --- |
| #jobmanager.rpc.address: bigdata11  high-availability:zookeeper //73行  #指定高可用模式（必须） //88行  high-availability.zookeeper.quorum:bigdata11:2181,bigdata12:2181,bigdata13:2181  #ZooKeeper仲裁是ZooKeeper服务器的复制组，它提供分布式协调服务（必须） //82行  high-availability.storageDir:hdfs:///flink/ha/  #JobManager元数据保存在文件系统storageDir中，只有指向此状态的指针存储在ZooKeeper中（必须） //没有  high-availability.zookeeper.path.root:/flink  #根ZooKeeper节点，在该节点下放置所有集群节点（推荐） //没有  high-availability.cluster-id:/flinkCluster  ＃自定义集群（推荐）  state.backend: filesystem  state.checkpoints.dir: hdfs:///flink/checkpoints  state.savepoints.dir: hdfs:///flink/checkpoints |

修改conf/zoo.cfg

|  |
| --- |
| server.1=bigdata11:2888:3888  server.2=bigdata12:2888:3888  server.3=bigdata13:2888:3888 |

修改conf/masters

|  |
| --- |
| bigdata11:8081  bigdata12:8081 |

修改slaves

|  |
| --- |
| bigdata12  bigdata13 |

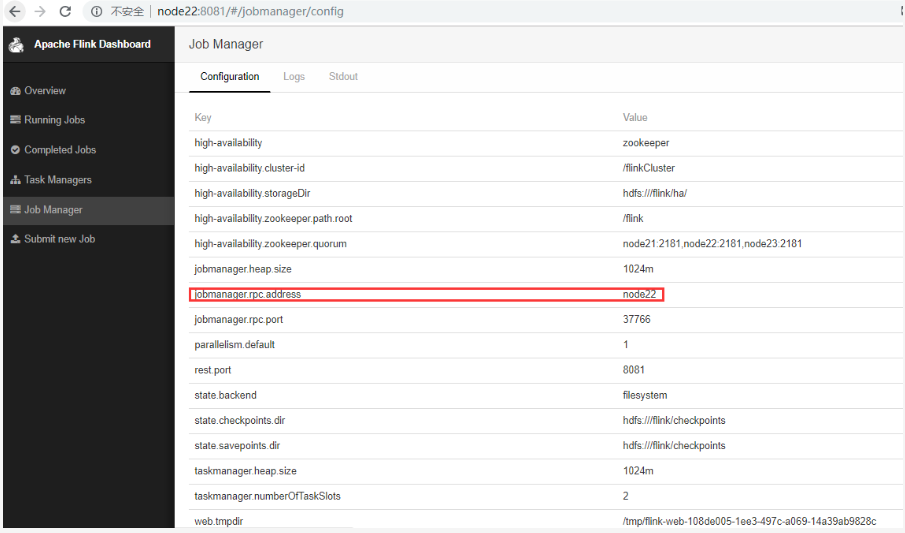
同步配置文件conf到各节点

**2） 启动HA**

先启动zookeeper集群各节点（测试环境中也可以用Flink自带的start-zookeeper-quorum.sh），启动dfs ,再启动flink

[itstar@bigdata11 flink-1.6.1]$ start-cluster.sh

WebUI查看，这是会自动产生一个主Master，如下



**3） 验证HA**

手动杀死bigdata12上的master，此时，bigdata11上的备用master转为主mater。

**4）手动将JobManager / TaskManager实例添加到群集**

您可以使用bin/jobmanager.sh和bin/taskmanager.sh脚本将JobManager和TaskManager实例添加到正在运行的集群中。

添加JobManager

bin/jobmanager.sh ((start|start-foreground) [host] [webui-port])|stop|stop-all

添加TaskManager

bin/taskmanager.sh start|start-foreground|stop|stop-all

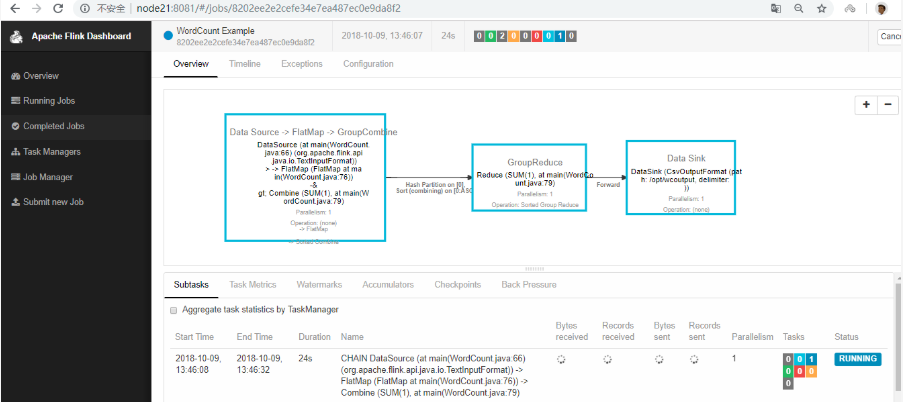
[itstar@bigdata12 flink-1.6.1]$ jobmanager.sh start bigdata12

新添加的为从master。

**9. 运行测试任务**

[itstar@bigdata11 flink-1.6.1]$ flink run -m bigdata11:8081 ./examples/batch/WordCount.jar --input /opt/wcinput/wc.txt --output /opt/wcoutput/

[itstar@bigdata11 flink-1.6.1]$ bin/flink run -m bigdata11:8081 ./examples/batch/WordCount.jar --input hdfs:///emp.csv --output hdfs:///user/itstar/output2



## （4） Yarn Cluster模式

**1. 引入**

在一个企业中，为了最大化的利用集群资源，一般都会在一个集群中同时运行多种类型的 Workload。因此 Flink 也支持在 Yarn 上面运行。首先，让我们了解下 Yarn 和 Flink 的关系。

在图中可以看出，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（也就是 App Master）和 TaskManager。

启动新的Flink YARN会话时，客户端首先检查所请求的资源（容器和内存）是否可用。之后，它将包含Flink和配置的jar上传到HDFS（步骤1）。

客户端的下一步是请求（步骤2）YARN容器以启动*ApplicationMaster*（步骤3）。由于客户端将配置和jar文件注册为容器的资源，因此在该特定机器上运行的YARN的NodeManager将负责准备容器（例如，下载文件）。完成后，将启动*ApplicationMaster*（AM）。

该*JobManager*和AM在同一容器中运行。一旦它们成功启动，AM就知道JobManager（它自己的主机）的地址。它正在为TaskManagers生成一个新的Flink配置文件（以便它们可以连接到JobManager）。该文件也上传到HDFS。此外，*AM*容器还提供Flink的Web界面。YARN代码分配的所有端口都是*临时端口*。这允许用户并行执行多个Flink YARN会话。

之后，AM开始为Flink的TaskManagers分配容器，这将从HDFS下载jar文件和修改后的配置。完成这些步骤后，即可建立Flink并准备接受作业。

**2. 修改环境变量**

export  HADOOP\_CONF\_DIR= /opt/module/hadoop-2.7.6/etc/hadoop

**3. 部署启动**

[itstar@bigdata11 flink-1.6.1]$ yarn-session.sh -d -s 1 -tm 800 -n 2

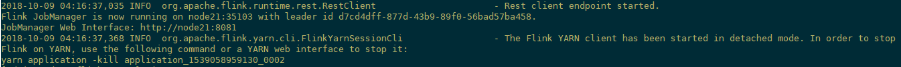
-n : TaskManager的数量，相当于executor的数量

-s : 每个JobManager的core的数量，executor-cores。建议将slot的数量设置每台机器的处理器数量

-tm : 每个TaskManager的内存大小，executor-memory

-jm : JobManager的内存大小，driver-memory

上面的命令的意思是，同时向Yarn申请3个container，其中 2 个 Container 启动 TaskManager（-n 2），每个 TaskManager 拥有两个 Task Slot（-s 2），并且向每个 TaskManager 的 Container 申请 800M 的内存，以及一个ApplicationMaster（Job Manager）。



Flink部署到Yarn Cluster后，会显示Job Manager的连接细节信息。

Flink on Yarn会覆盖下面几个参数，如果不希望改变配置文件中的参数，可以动态的通过-D选项指定，如 -Dfs.overwrite-files=true -Dtaskmanager.network.numberOfBuffers=16368

jobmanager.rpc.address：因为JobManager会经常分配到不同的机器上

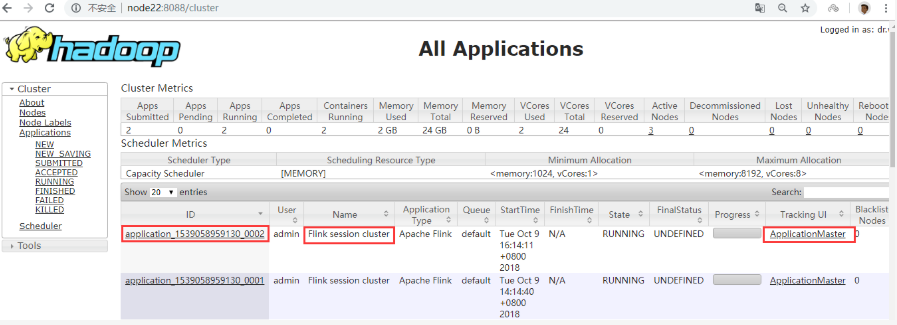
taskmanager.tmp.dirs：使用Yarn提供的tmp目录

parallelism.default：如果有指定slot个数的情况下

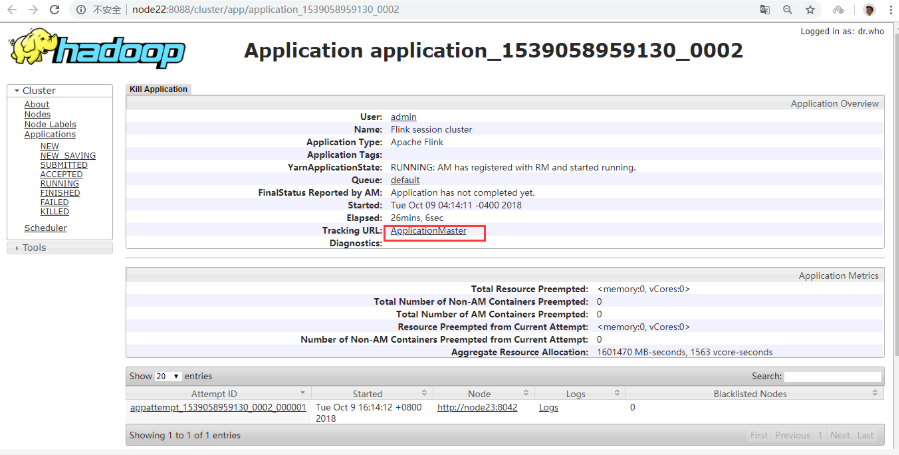
yarn-session.sh会挂起进程，所以可以通过在终端使用CTRL+C或输入stop停止yarn-session。

如果不希望Flink Yarn client长期运行，Flink提供了一种detached YARN session，启动时候加上参数-d或—detached

在上面的命令成功后，我们就可以在 Yarn Application 页面看到 Flink 的纪录。如下图。



如果在虚拟机中测试，可能会遇到错误。这里需要注意内存的大小，Flink 向 Yarn 会申请多个 Container，但是 Yarn 的配置可能限制了 Container 所能申请的内存大小，甚至 Yarn 本身所管理的内存就很小。这样很可能无法正常启动 TaskManager，尤其当指定多个 TaskManager 的时候。因此，在启动 Flink 之后，需要去 Flink 的页面中检查下 Flink 的状态。这里可以从 RM 的页面中，直接跳转（点击 Tracking UI）。这时候 Flink 的页面如图



yarn-session.sh启动命令参数如下：

[itstar@bigdata11 flink-1.6.1]$ yarn-session.sh --help

Usage:

Required

-n,--container <arg> Number of YARN container to allocate (=Number of Task Managers)

Optional

-D <property=value> use value for given property

-d,--detached If present, runs the job in detached mode

-h,--help Help for the Yarn session CLI.

-id,--applicationId <arg> Attach to running YARN session

-j,--jar <arg> Path to Flink jar file

-jm,--jobManagerMemory <arg> Memory for JobManager Container with optional unit (default: MB)

-m,--jobmanager <arg> Address of the JobManager (master) to which to connect. Use this flag to connect to a different JobManager than the one specified i

n the configuration. -n,--container <arg> Number of YARN container to allocate (=Number of Task Managers)

-nl,--nodeLabel <arg> Specify YARN node label for the YARN application

-nm,--name <arg> Set a custom name for the application on YARN

-q,--query Display available YARN resources (memory, cores)

-qu,--queue <arg> Specify YARN queue.

-s,--slots <arg> Number of slots per TaskManager

-st,--streaming Start Flink in streaming mode

-t,--ship <arg> Ship files in the specified directory (t for transfer)

-tm,--taskManagerMemory <arg> Memory per TaskManager Container with optional unit (default: MB)

-yd,--yarndetached If present, runs the job in detached mode (deprecated; use non-YARN specific option instead)

-z,--zookeeperNamespace <arg> Namespace to create the Zookeeper sub-paths for high availability mode

**4. 提交任务**

之后，我们可以通过这种方式提交我们的任务

[itstar@bigdata11 flink-1.6.1]$ ./bin/flink run -m yarn-cluster -yn 2 ./examples/batch/WordCount.jar --input /opt/wcinput/wc.txt --output /opt/wcoutput/

bin/flink run -m yarn-cluster -yn2 examples/batch/WordCount.jar --input /input/ --output /Andy

以上命令在参数前加上y前缀，-yn表示TaskManager个数。

在这个模式下，同样可以使用-m yarn-cluster提交一个"运行后即焚"的detached yarn（-yd）作业到yarn cluster。

**5. 停止yarn cluster**

yarn application -kill application\_1539058959130\_0001

**6. Yarn模式的HA**

应用最大尝试次数（yarn-site.xml），您必须配置为尝试应用的最大数量的设置yarn-site.xml，当前YARN版本的默认值为2（表示允许单个JobManager失败）。

<property>

<name>yarn.resourcemanager.am.max-attempts</name>

<value>4</value>

<description>The maximum number of application master execution attempts</description>

</property>

申请尝试（flink-conf.yaml），您还必须配置最大尝试次数conf/flink-conf.yaml： yarn.application-attempts：10

示例：高度可用的YARN会话

配置HA模式和zookeeper法定人数在conf/flink-conf.yaml：

high-availability: zookeeper

high-availability.zookeeper.quorum: bigdata11:2181,bigdata12:2181,bigdata13:2181

high-availability.storageDir: hdfs:///flink/recovery

high-availability.zookeeper.path.root: /flink

yarn.application-attempts: 10

配置ZooKeeper的服务器中conf/zoo.cfg（目前它只是可以运行每台机器的单一的ZooKeeper服务器）：

server.1=bigdata11:2888:3888

server.2=bigdata12:2888:3888

server.3=bigdata13:2888:3888

启动ZooKeeper仲裁：

$ bin / start-zookeeper-quorum.sh

启动HA群集：

$ bin / yarn-session.sh -n 2

# [（五）Flink开发IDEA环境搭建与测试](https://www.cnblogs.com/frankdeng/p/9760015.html)

## （1）IDEA开发环境

先虚拟机联网，然后执行yum -y install nc

nc是用来打开端口的工具

然后nc -l 9000

**1.pom文件设置**

|  |
| --- |
| <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <encoding>UTF-8</encoding>  <scala.version>2.11.12</scala.version>  <scala.binary.version>2.11</scala.binary.version>  <hadoop.version>2.8.4</hadoop.version>  <flink.version>1.6.1</flink.version>  </properties>  <dependencies>  <dependency>  <groupId>org.scala-lang</groupId>  <artifactId>scala-library</artifactId>  <version>${scala.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-java</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-streaming-java\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-streaming-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-clients\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka-0.10\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.hadoop</groupId>  <artifactId>hadoop-client</artifactId>  <version>${hadoop.version}</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.38</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.22</version>  </dependency>  </dependencies>  <build>  <sourceDirectory>src/main/scala</sourceDirectory>  <testSourceDirectory>src/test/scala</testSourceDirectory>  <plugins>  <plugin>  <groupId>net.alchim31.maven</groupId>  <artifactId>scala-maven-plugin</artifactId>  <version>3.2.0</version>  <executions>  <execution>  <goals>  <goal>compile</goal>  <goal>testCompile</goal>  </goals>  <configuration>  <args>  <!-- <arg>-make:transitive</arg> -->  <arg>-dependencyfile</arg>  <arg>${project.build.directory}/.scala\_dependencies</arg>  </args>  </configuration>  </execution>  </executions>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.18.1</version>  <configuration>  <useFile>false</useFile>  <disableXmlReport>true</disableXmlReport>  <includes>  <include>\*\*/\*Test.\*</include>  <include>\*\*/\*Suite.\*</include>  </includes>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-shade-plugin</artifactId>  <version>3.0.0</version>  <executions>  <execution>  <phase>package</phase>  <goals>  <goal>shade</goal>  </goals>  <configuration>  <filters>  <filter>  <artifact>\*:\*</artifact>  <excludes>  <exclude>META-INF/\*.SF</exclude>  <exclude>META-INF/\*.DSA</exclude>  <exclude>META-INF/\*.RSA</exclude>  </excludes>  </filter>  </filters>  <transformers>  <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">  <mainClass>org.apache.spark.WordCount</mainClass>  </transformer>  </transformers>  </configuration>  </execution>  </executions>  </plugin>  </plugins>  </build> |

**2.flink开发流程**

Flink具有特殊类DataSet并DataStream在程序中表示数据。您可以将它们视为可以包含重复项的不可变数据集合。在DataSet数据有限的情况下，对于一个DataStream元素的数量可以是无界的。

这些集合在某些关键方面与常规Java集合不同。首先，它们是不可变的，这意味着一旦创建它们就无法添加或删除元素。你也不能简单地检查里面的元素。

集合最初通过在弗林克程序添加源创建和新的集合从这些通过将它们使用API方法如衍生map，filter等等。

Flink程序看起来像是转换数据集合的常规程序。每个程序包含相同的基本部分：

1.获取execution environment,

final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

2.加载/创建初始化数据

DataStream<String> text = env.readTextFile("file:///path/to/file");

3.指定此数据的转换

**val mapped = input.map { x => x.toInt }**

4.指定放置计算结果的位置

**writeAsText(String path)**

**print()**

5.触发程序执行

在local模式下执行程序

execute()

将程序达成jar运行在线上

./bin/flink run \

-m node21:8081 \

./examples/batch/WordCount.jar \

--input  hdfs:///user/itstar/input/wc.txt \

--output  hdfs:///user/itstar/output2  \

## （2） Wordcount案例

**1.Scala代码**

|  |
| --- |
| import org.apache.flink.api.java.utils.ParameterTool  import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment  import org.apache.flink.streaming.api.windowing.time.Time  object SocketWindowWordCountScala {  def main(args: Array[String]) : Unit = {  // 定义一个数据类型保存单词出现的次数  case class WordWithCount(word: String, count: Long)  // port 表示需要连接的端口  val port: Int = try {  ParameterTool.fromArgs(args).getInt("port")  } catch {  case e: Exception => {  System.err.println("No port specified. Please run 'SocketWindowWordCount --port <port>'")  return  }  }  // 获取运行环境  val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment  // 连接此socket获取输入数据  val text = env.socketTextStream("node21", port, '\n')  //需要加上这一行隐式转换 否则在调用flatmap方法的时候会报错  import org.apache.flink.api.scala.\_  // 解析数据, 分组, 窗口化, 并且聚合求SUM  val windowCounts = text  .flatMap { w => w.split("\\s") }  .map { w => WordWithCount(w, 1) }  .keyBy("word")  .timeWindow(Time.seconds(5), Time.seconds(1))  .sum("count")  // 打印输出并设置使用一个并行度  windowCounts.print().setParallelism(1)  env.execute("Socket Window WordCount")  }  } |

**2.Java代码**

|  |
| --- |
| import org.apache.flink.api.common.functions.FlatMapFunction;  import org.apache.flink.api.java.utils.ParameterTool;  import org.apache.flink.streaming.api.datastream.DataStream;  import org.apache.flink.streaming.api.datastream.DataStreamSource;  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  import org.apache.flink.streaming.api.windowing.time.Time;  import org.apache.flink.util.Collector;  public class WordCount {  //先在虚拟机上打开你的端口号 nc -l 9000  public static void main(String[] args) throws Exception {  //定义socket的端口号  int port;  try{  ParameterTool parameterTool = ParameterTool.fromArgs(args);  port = parameterTool.getInt("port");  }catch (Exception e){  System.err.println("没有指定port参数，使用默认值9000");  port = 9000;  }  //获取运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //连接socket获取输入的数据  DataStreamSource<String> text = env.socketTextStream("192.168.1.52", port, "\n");  //计算数据  DataStream<WordWithCount> windowCount = text.flatMap(new FlatMapFunction<String, WordWithCount>() {  public void flatMap(String value, Collector<WordWithCount> out) throws Exception {  String[] splits = value.split("\\s");  for (String word:splits) {  out.collect(new WordWithCount(word,1L));  }  }  })//打平操作，把每行的单词转为<word,count>类型的数据  .keyBy("word")//针对相同的word数据进行分组  .timeWindow(Time.seconds(2),Time.seconds(1))//指定计算数据的窗口大小和滑动窗口大小  .sum("count");  //把数据打印到控制台  windowCount.print()  .setParallelism(1);//使用一个并行度  //注意：因为flink是懒加载的，所以必须调用execute方法，上面的代码才会执行  env.execute("streaming word count");  }  /\*\*  \* 主要为了存储单词以及单词出现的次数  \*/  public static class WordWithCount{  public String word;  public long count;  public WordWithCount(){}  public WordWithCount(String word, long count) {  this.word = word;  this.count = count;  }  @Override  public String toString() {  return "WordWithCount{" +  "word='" + word + '\'' +  ", count=" + count +  '}';  }  }  } |

**3.运行测试**

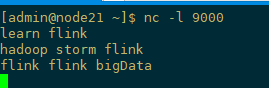
首先，使用nc命令启动一个本地监听，命令是：

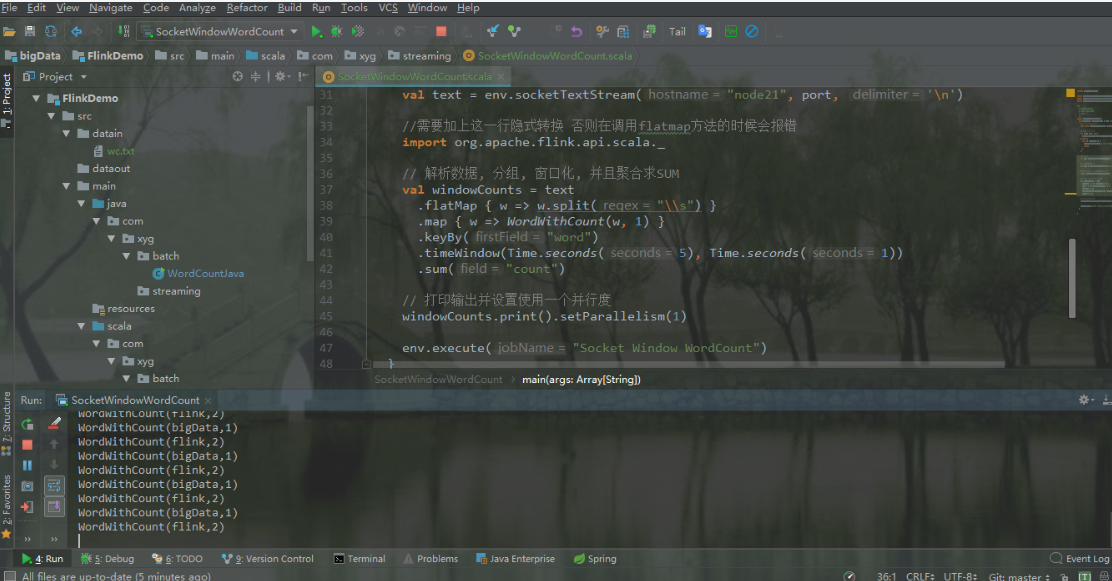
[itstar@node21 ~]$ nc -l 9000

通过netstat命令观察9000端口。 netstat -anlp | grep 9000，启动监听如果报错：-bash: nc: command not found，请先安装nc，在线安装命令：yum -y install nc。

然后，IDEA上运行flink官方案例程序

node21上输入



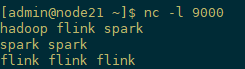


**4.集群测试**

**这里单机测试官方案例**

|  |
| --- |
| [itstar@node21 flink-1.6.1]$ pwd  /opt/flink-1.6.1  [itstar@node21 flink-1.6.1]$ ./bin/start-cluster.sh  Starting cluster.  Starting standalonesession daemon on host node21.  Starting taskexecutor daemon on host node21.  [itstar@node21 flink-1.6.1]$ jps  StandaloneSessionClusterEntrypoint  TaskManagerRunner  Jps  [itstar@node21 flink-1.6.1]$ ./bin/flink run examples/streaming/SocketWindowWordCount.jar --port 9000 |

单词在5秒的时间窗口（处理时间，翻滚窗口）中计算并打印到stdout。监视TaskManager的输出文件并写入一些文本nc（输入在点击后逐行发送到Flink）：



## （3）使用IDEA开发离线程序

Dataset是flink的常用程序，数据集通过source进行初始化，例如读取文件或者序列化集合，然后通过transformation（filtering、mapping、joining、grouping）将数据集转成，然后通过sink进行存储，既可以写入hdfs这种分布式文件系统，也可以打印控制台，flink可以有很多种运行方式，如local、flink集群、yarn等.

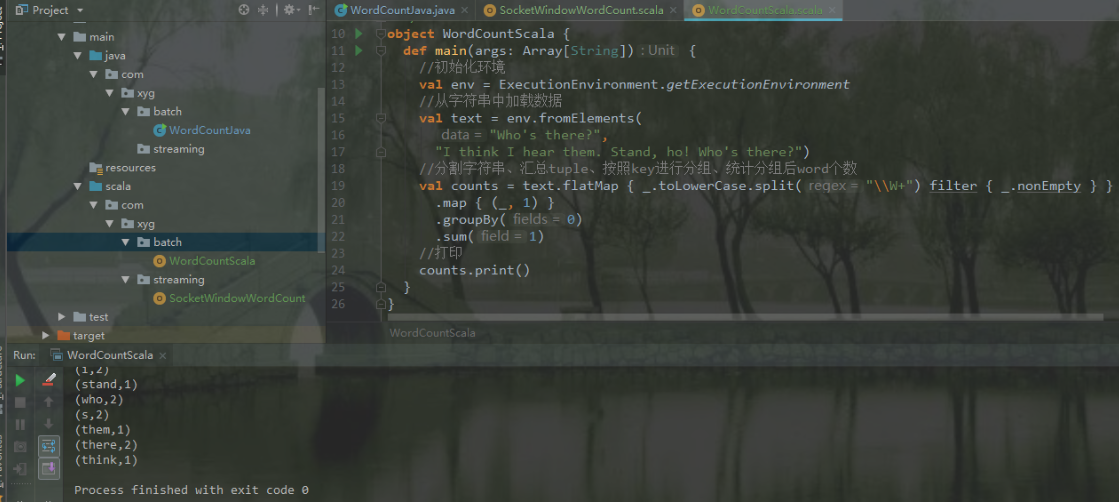
**1. scala程序**

|  |
| --- |
| import org.apache.flink.api.scala.ExecutionEnvironment  import org.apache.flink.api.scala.\_  object WordCountScala{  def main(args: Array[String]) {  //初始化环境  val env = ExecutionEnvironment.getExecutionEnvironment  //从字符串中加载数据  val text = env.fromElements(  "Who's there?",  "I think I hear them. Stand, ho! Who's there?")  //分割字符串、汇总tuple、按照key进行分组、统计分组后word个数  val counts = text.flatMap { \_.toLowerCase.split("\\W+") filter { \_.nonEmpty } }  .map { (\_, 1) }  .groupBy(0)  .sum(1)  //打印  counts.print()  }  } |

**2. java程序**

|  |
| --- |
| import org.apache.flink.api.common.functions.FlatMapFunction;  import org.apache.flink.api.java.DataSet;  import org.apache.flink.api.java.ExecutionEnvironment;  import org.apache.flink.api.java.tuple.Tuple2;  import org.apache.flink.util.Collector;  public class WordCountJava {  public static void main(String[] args) throws Exception {  //构建环境  final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  //通过字符串构建数据集  DataSet<String> text = env.fromElements(  "Who's there?",  "I think I hear them. Stand, ho! Who's there?");  //分割字符串、按照key进行分组、统计相同的key个数  DataSet<Tuple2<String, Integer>> wordCounts = text  .flatMap(new LineSplitter())  .groupBy(0)  .sum(1);  //打印  wordCounts.print();  }  //分割字符串的方法  public static class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {  @Override  public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {  for (String word : line.split(" ")) {  out.collect(new Tuple2<String, Integer>(word, 1));  }  }  }  } |

**3.运行**



# [（六）监控](https://www.cnblogs.com/frankdeng/p/9760015.html)维基百科的编辑日志

## （1）pom.xml

|  |
| --- |
| <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <encoding>UTF-8</encoding>  <scala.version>2.11.12</scala.version>  <scala.binary.version>2.11</scala.binary.version>  <hadoop.version>2.8.4</hadoop.version>  <flink.version>1.6.1</flink.version>  </properties>  <dependencies>  <dependency>  <groupId>org.scala-lang</groupId>  <artifactId>scala-library</artifactId>  <version>${scala.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-java</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-streaming-java\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-streaming-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-clients\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka-0.10\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.hadoop</groupId>  <artifactId>hadoop-client</artifactId>  <version>${hadoop.version}</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.38</version>  </dependency>  <!--<dependency>-->  <!--<groupId>com.alibaba</groupId>-->  <!--<artifactId>fastjson</artifactId>-->  <!--<version>1.2.22</version>-->  <!--</dependency>-->  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-wikiedits\_2.11</artifactId>  <version>1.6.1</version>  </dependency>  </dependencies> |

## （2）代码

|  |
| --- |
| import org.apache.flink.api.common.functions.FoldFunction;  import org.apache.flink.api.java.functions.KeySelector;  import org.apache.flink.api.java.tuple.Tuple2;  import org.apache.flink.streaming.api.datastream.DataStream;  import org.apache.flink.streaming.api.datastream.KeyedStream;  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  import org.apache.flink.streaming.api.windowing.time.Time;  import org.apache.flink.streaming.connectors.wikiedits.WikipediaEditEvent;  import org.apache.flink.streaming.connectors.wikiedits.WikipediaEditsSource;  public class WikipediaAnalysis {  public static void main(String[] args) throws Exception {  //创建一个streaming程序运行的上下文  StreamExecutionEnvironment see = StreamExecutionEnvironment.getExecutionEnvironment();  //sowurce部分---数据来源部分  DataStream<WikipediaEditEvent> edits = see.addSource(new WikipediaEditsSource());  KeyedStream<WikipediaEditEvent, String> keyedEdits = edits  .keyBy(new KeySelector<WikipediaEditEvent, String>() {  @Override  public String getKey(WikipediaEditEvent event) {  return event.getUser();  }  });  DataStream<Tuple2<String, Long>> result = keyedEdits  .timeWindow(Time.seconds(5))  .fold(new Tuple2<>("", 0L), new FoldFunction<WikipediaEditEvent, Tuple2<String, Long>>() {  @Override  public Tuple2<String, Long> fold(Tuple2<String, Long> acc, WikipediaEditEvent event) {  acc.f0 = event.getUser();  acc.f1 += event.getByteDiff();  return acc;  }  });  result.print();  see.execute();  }  } |

博客：

<https://www.jianshu.com/p/16323566f3c6>

<http://zhuanlan.51cto.com/art/201809/584104.htm>