Flink超神文档

Flink源码(GitHub):

- git@github.com:bjmashibing/Flink-Study.git
- https://github.com/bjmashibing/Flink-Study

Flink初次见面

什么是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的世界观是数据流

对 Flink 而言，其所要处理的主要场景就是流数据，批数据只是流数据的一个极限特例而已，所以 Flink 也是一款真正的流批统一的计算引擎

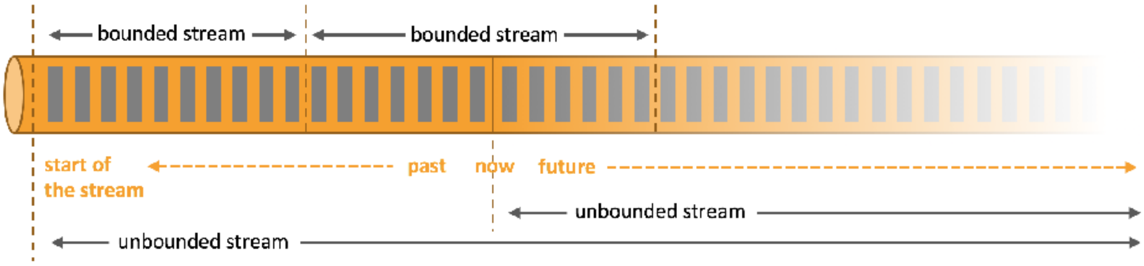
什么是Unbounded streams?

无界流 有定义流的开始，但没有定义流的结束。它们会无休止地产生数据。无界流的数据必须持续处理，即数据被摄取后需要立刻处理。我们不能等到所有数据都到达再处理，因为输入是无限的，在任何时候输入都不会完成。处理无界数据通常要求以特定顺序摄取事件，例如事件发生的顺序，以便能够推断结果的完整性

什么是Bounded streams?

有界流 有定义流的开始，也有定义流的结束。有界流可以在摄取所有数据后再进行计算。有界流所有数据可以被排序，所以并不需要有序摄取。有界流处理通常被称为批处理

一图秒懂：无界流与有界流

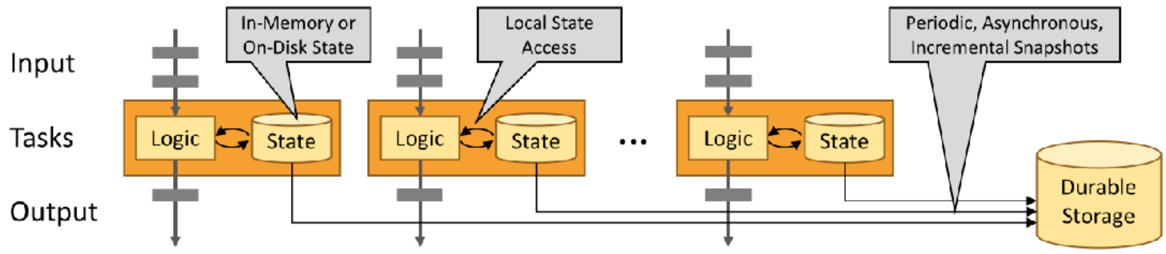


什么是stateful computations?

有状态的计算：每次进行数据计算的时候基于之前数据的计算结果（状态）做计算，并且每次计算结果都会保存到存储介质中，计算关联上下文context

基于有状态的计算不需要将历史数据重新计算，提高了计算效率

无状态的计算：每次进行数据计算只是考虑当前数据，不会使用之前数据的计算结果



Flink使用用户

自 2019 年 1 月起，阿里巴巴逐步将内部维护的 Blink 回馈给 Flink 开源社区，目前贡献代码数量已超过 100 万行。国内包括腾讯、百度、字节跳动等公司，国外包括 Uber、Netflix 等公司都是 Flink 的使用者

Flink的特点和优势

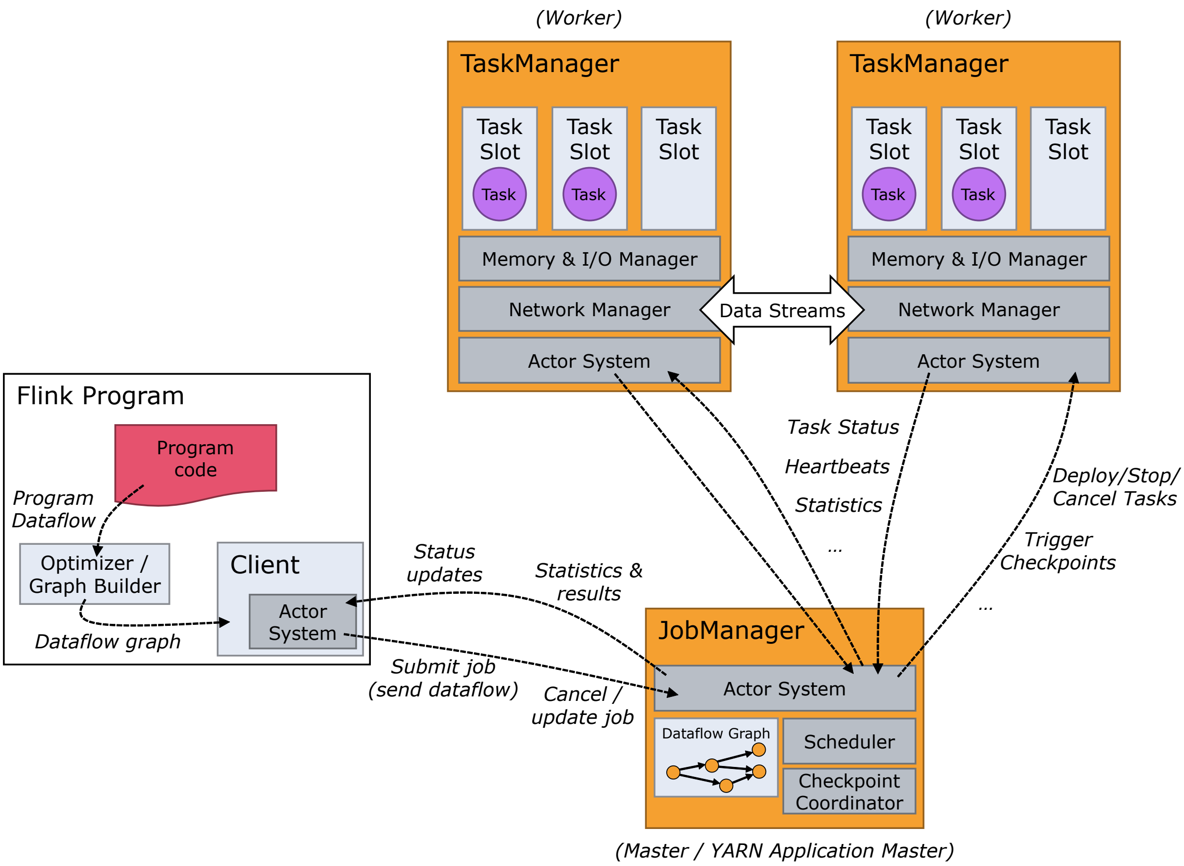
1、同时支持高吞吐、低延迟、高性能
2、支持事件时间（Event Time）概念，结合Watermark处理乱序数据
3、支持有状态计算，并且支持多种状态 内存、文件、RocksDB
4、支持高度灵活的窗口（Window）操作 time、count、session
5、基于轻量级分布式快照（CheckPoint）实现的容错 保证exactly-once语义

6、基于JVM实现独立的内存管理
7、Save Points（保存点）

Flink安装&部署

Flink基本架构

Flink系统架构中包含了两个角色，分别是JobManager和TaskManager，是一个典型的Master-Slave架构。JobManager相当于是Master，TaskManager相当于是Slave



JobManager（JVM进程）作用

JobManager负责整个集群的资源管理与任务管理，在一个集群中只能由一个正在工作（active）的JobManager，如果HA集群，那么其他JobManager一定是standby状态

（1）资源调度

- 集群启动，TaskManager会将当前节点的资源信息注册给JobManager，所有TaskManager全部注册完毕，集群启动成功，此时JobManager就掌握整个集群的资源情况
- client提交Application给JobManager，JobManager会根据集群中的资源情况，为当前的Application分配TaskSlot资源

（2）任务调度

- 根据各个TaskManager节点上的资源分发task到TaskSlot中运行
- Job执行过程中，JobManager会根据设置的触发策略触发checkpoint，通知TaskManager开始checkpoint
- 任务执行完毕，JobManager会将Job执行的信息反馈给client，并且释放TaskManager资源

TaskManager（JVM进程）作用

- 负责当前节点上的任务运行及当前节点上的资源管理，TaskManager资源通过TaskSlot进行了划分，每个TaskSlot代表的是一份固定资源。例如，具有三个 slots 的 TaskManager 会将其管理的内存资源分成三等份给每个 slot。 划分资源意味着 subtask 之间不会竞争内存资源，但是也意味着它们只拥有固定的资源。注意这里并没有 CPU 隔离，当前 slots 之间只是划分任务的内存资源
- 负责TaskManager之间的数据交换

---

client客户端

负责将当前的任务提交给JobManager，提交任务的常用方式：命令提交、web页面提交。获取任务的执行信息

Standalone集群安装&测试

Standalone是独立部署模式，它不依赖其他平台，不依赖任何的资源调度框架

Standalone集群是由JobManager、TaskManager两个JVM进程组成

集群角色划分

node01	node02	node03	node04
JobManager	TaskManager	TaskManager	TaskManager

安装步骤

1. 官网下载Flink安装包

   Apache Flink® 1.10.0 is our latest stable release.现在最稳定的是1.10.0，不建议采用这个版本，刚从1.9升级到1.10，会存在一些bug，不建议采用小版本号为0的安装包，所以我们建议使用1.9.2版本

   下载链接:https://mirrors.tuna.tsinghua.edu.cn/apache/flink/flink-1.9.2/flink-1.9.2-bin-scala_2.11.tgz

2. 安装包上传到node01节点

3. 解压、修改配置文件

   解压：tar -zxf flink-1.9.2-bin-scala_2.11.tgz

   修改flink-conf.yaml配置文件

```
jobmanager.rpc.address: node01  JobManager地址
jobmanager.rpc.port: 6123       JobManagerRPC通信端口
jobmanager.heap.size: 1024m     JobManager所能使用的堆内存大小
taskmanager.heap.size: 1024m    TaskManager所能使用的堆内存大小
taskmanager.numberOfTaskSlots: 2 TaskManager管理的TaskSlot个数，依据当前物理机的核心数来配置，一般预留出一部分核心（25%）给系统及其他进程使用，一个slot对应一个core。如果core支持超线程，那么slot个数*2
rest.port: 8081                 指定WebUI的访问端口
```

修改slaves配置文件

```
node02
node03
node04
```

4. 同步安装包到其他的节点

   同步到node02 scp -r flink-1.9.2 node02:`pwd`

   同步到node03 scp -r flink-1.9.2 node03:`pwd`

   同步到node04 scp -r flink-1.9.2 node04:`pwd`

5. node01配置环境变量

```
vim ~/.bashrc
export FLINK_HOME=/opt/software/flink/flink-1.9.2
export PATH=$PATH:$FLINK_HOME/bin
source ~/.bashrc
```

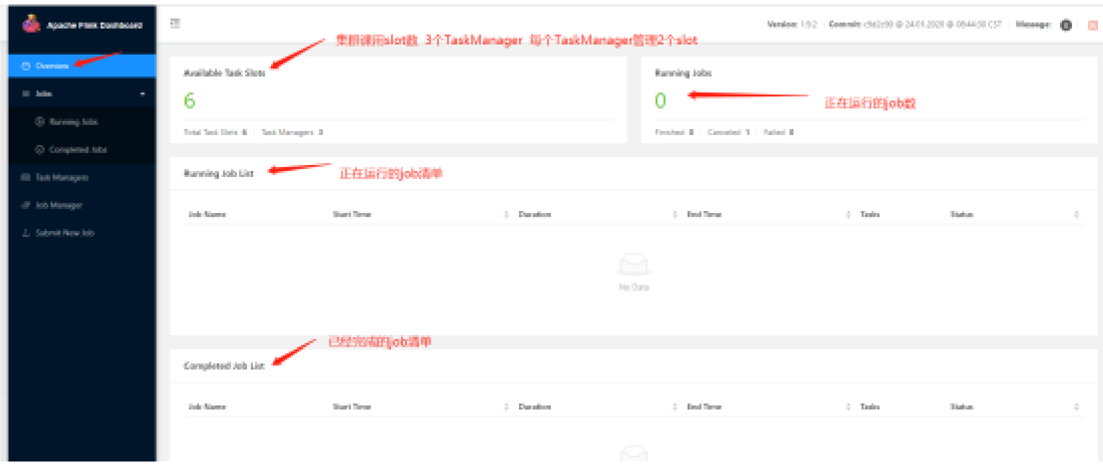
6. 启动standalone集群

   启动集群：start-cluster.sh

   关闭集群：stop-cluster.sh

7. 查看Flink Web UI页面

   http://node01:8081/ 可通过rest.port参数自定义端口



提交Job到standalone集群

常用提交任务的方式有两种，分别是命令提交和Web页面提交

1. **命令提交：**

   flink run -c com.msb.stream.WordCount StudyFlink-1.0-SNAPSHOT.jar

   -c 指定主类

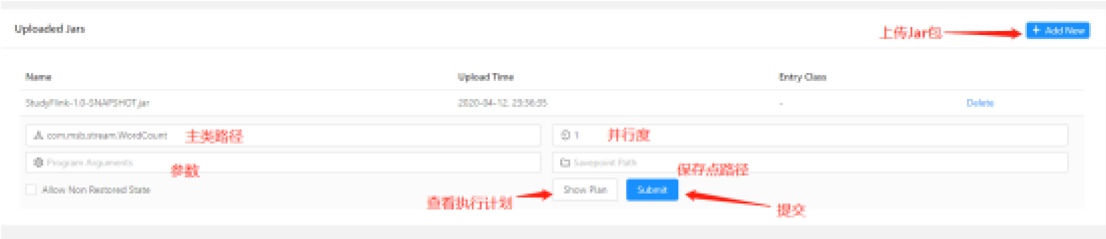
   -d 独立运行、后台运行

   -p 指定并行度

2. **Web页面提交：**

   在Web中指定Jar包的位置、主类路径、并行数等

   web.submit.enable: true一定是true，否则不支持Web提交Application



3. 启动scala-shell测试

```
start-scala-shell.sh remote <hostname> <portnumber>
```

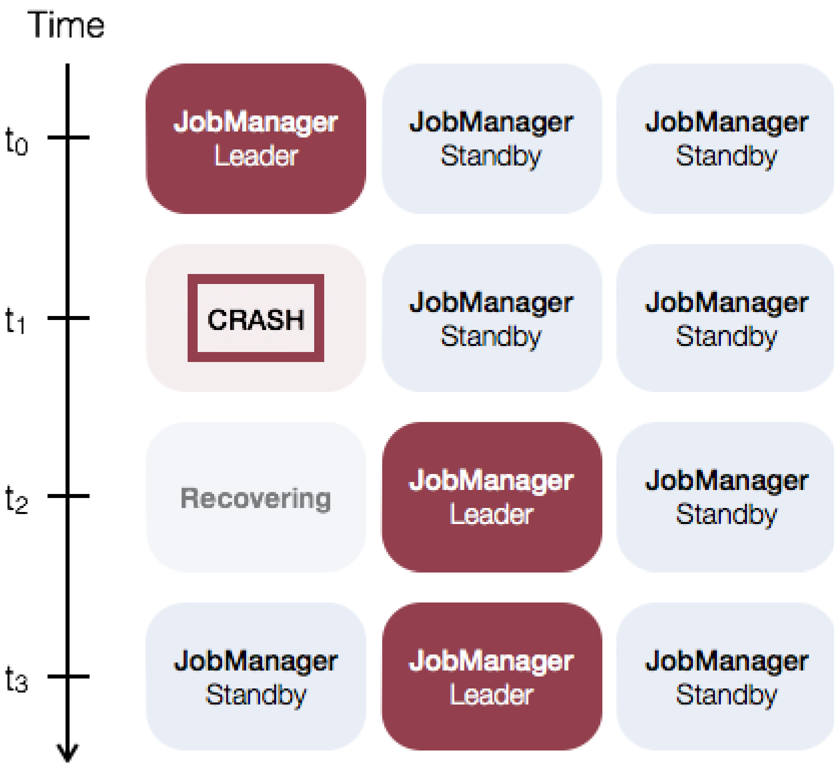
Standalone HA集群安装&测试

JobManager协调每个flink任务部署,它负责调度和资源管理

默认情况下，每个flink集群只有一个JobManager，这将导致一个单点故障(SPOF single-point-of-failure)：如果JobManager挂了，则不能提交新的任务，并且运行中的程序也会失败。

使用JobManager HA，集群可以从JobManager故障中恢复，从而避免SPOF

Standalone模式（独立模式）下JobManager的高可用性的基本思想是，任何时候都有一个 Active JobManager，并且多个Standby JobManagers。 Standby JobManagers可以在Master JobManager 挂掉的情况下接管集群成为Master JobManager。 这样保证了没有单点故障，一旦某一个Standby JobManager接管集群，程序就可以继续运行。 Standby JobManager和Active JobManager实例之间没有明确区别。 每个JobManager可以成为Active或Standby节点



如何单独启动JobManager jobmanager.sh

如何单独启动TaskManager taskmanager.sh

集群角色划分

	node01	node02	node03	node04
JobManager	√	√	×	×
TaskManager	×	√	√	√

安装步骤

1. 修改配置文件conf/flink-conf.yaml

```
high-availability: zookeeper
high-availability.storageDir: hdfs://node01:9000/flink/ha/ 保存JobManager恢复所需要的所有元数据信息
high-availability.zookeeper.quorum: node01:2181,node02:2181,node03:2181 zookeeper地址
```

2. 修改配置文件conf/masters

```
node01:8081
node02:8081
```

3. 同步文件到各个节点
4. 下载支持Hadoop插件并且拷贝到各个节点的安装包的lib目录下

   下载地址：https://repo.maven.apache.org/maven2/org/apache/flink/flink-shaded-hadoop-2-uber/2.6.5-10.0/flink-shaded-hadoop-2-uber-2.6.5-10.0.jar

- HA集群测试

  http://node01:8081/

  http://node02:8081/

  两个页面一模一样 存在bug

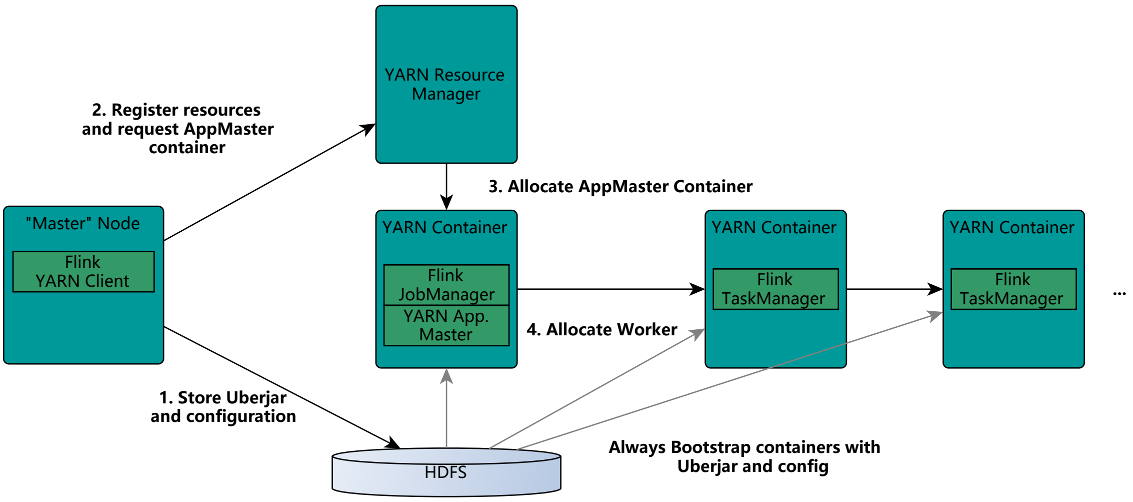
Flink on Yarn

Flink on Yarn是依托Yarn资源管理器，现在很多分布式任务都可以支持基于Yarn运行，这是在企业中使用最多的方式。Why?

（1）基于Yarn的运行模式可以充分使用集群资源，Spark on Yarn、MapReduce on Yarn、Flink on Yarn等 多套计算框架都可以基于Yarn运行，充分利用集群资源

（2）基于Yarn的运行模式降低维护成本

运行流程



1. 每当创建一个新flink的yarn session的时候，客户端会首先检查要请求的资源(containers和memory)是否可用。然后，将包含flink相关的jar包盒配置上传到HDFS
2. 客户端会向ResourceManager申请一个yarn container 用以启动ApplicationMaster。由于客户端已经将配置和jar文件上传到HDFS，ApplicationMaster将会下载这些jar和配置，然后启动成功
3. JobManager和AM运行于同一个container
4. AM开始申请启动Flink TaskManager的containers，这些container会从HDFS上下载jar文件和已修改的配置文件。一旦这些步骤完成，flink就可以接受任务了

Flink on Yarn两种运行模式

解脱了JobManager的压力 RM做资源管理 JobManager只负责任务管理

- yarn seesion(Start a long-running Flink cluster on YARN)这种方式是在yarn中先启动Flink集群，然后再提交作业，这个Flink集群一直停留再yarn中，一直占据了yarn集群的资源（只是JobManager会一直占用，没有Job运行TaskManager并不会运行），不管有没有任务运行。这种方式能够降低任务的启动时间
- Run a Flink job on YARN 每次提交一个Flink任务的时候，先去yarn中申请资源启动JobManager和TaskManager，然后在当前集群中运行，任务执行完毕，集群关闭。任务之间互相独立，互不影响，可以最大化的使用集群资源，但是每个任务的启动时间变长了

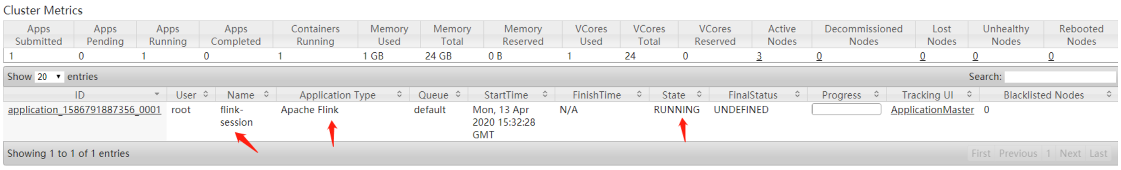
配置两种运行模式

yarn seesion模式配置

- Flink on Yarn依赖Yarn集群和HDFS集群，启动Yarn、HDFS集群 start-all.sh
- 下载支持Hadoop插件并且拷贝到各个节点的安装包的lib目录下

  下载地址：https://repo.maven.apache.org/maven2/org/apache/flink/flink-shaded-hadoop-2-uber/2.6.5-10.0/flink-shaded-hadoop-2-uber-2.6.5-10.0.jar
- 在yarn中启动Flink集群

```
启动：yarn-session.sh  -n 3 -s 3 -nm flink-session  -d -q
关闭：yarn application -kill applicationId

yarn-session选项：
-n,--container <arg>：在yarn中启动container的个数，实质就是TaskManager的个数
-s,--slots <arg>：每个TaskManager管理的Slot个数
-nm,--name <arg>:给当前的yarn-session(Flink集群)起一个名字
-d,--detached:后台独立模式启动，守护进程
-tm,--taskManagerMemory <arg>：TaskManager的内存大小 单位：MB
-jm,--jobManagerMemory <arg>：JobManager的内存大小 单位：MB
-q,--query：显示yarn集群可用资源（内存、core）
```



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	1	0	1	1 GB	24 GB	0 B	1	24	0	3	0	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes
application_1586791887356_0001	root	flink-session	Apache Flink	default	Mon, 13 Apr 2020 15:32:28 GMT	N/A	RUNNING	UNDEFINED		ApplicationMaster	0

Showing 1 to 1 of 1 entries First Previous 1 Next Last

- 提交Flink Job到yarn-session集群中运行

```
flink run -c com.msb.stream.WordCount -yid application_1586794520478_0007
~/StudyFlink-1.0-SNAPSHOT.jar

yid：指定yarn-session的ApplicationID
不使用yid也可以，因为在启动yarn-session的时候，在tmp临时目录下已经产生了一个隐藏小文件
[root@node01 bin]# vim /tmp/.yarn-properties-root
#Generated YARN properties file
#Mon Apr 13 23:39:43 CST 2020
parallelism=9
dynamicPropertiesString=
applicationID=application_1586791887356_0001
```

Run a Flink job on YARN模式配置

```
flink run -m yarn-cluster -yn 3 -ys 3 -ynm flink-job -c com.msb.stream.WordCount
~/StudyFlink-1.0-SNAPSHOT.jar

-yn,--container <arg> 表示分配容器的数量，也就是TaskManager的数量。
-d,--detached：设置在后台运行。
-yjm,--jobManagerMemory<arg>:设置JobManager的内存，单位是MB。
-ytm，--taskManagerMemory<arg>:设置每个TaskManager的内存，单位是MB。
-ynm,--name:给当前Flink application在Yarn上指定名称。
-yq,--query：显示yarn中可用的资源（内存、cpu核数）
-yqu,--queue<arg> :指定yarn资源队列
-ys,--slots<arg> :每个TaskManager使用的Slot数量。
```

Flink on YARN HA集群安装&测试

无论以什么样的模式提交Application到Yarn中运行，都会启动一个yarn-session(Flink 集群)，依然是由JobManager和TaskManager组成，那么JobManager节点如果宕机，那么整个Flink集群就不会正常运转，所以接下来搭建Flink on YARN HA集群

安装步骤

- 修改Hadoop安装包下的yarn-site.xml文件

```
<property>
  <name>yarn.resourcemanager.am.max-attempts</name>
  <value>10</value>
  <description>
   The maximum number of application master execution attempts AppMaster最大重试次数
  </description>
</property>
```

- 修改Flink安装包下的flin-conf.yaml文件

```
high-availability: zookeeper
high-availability.storageDir: hdfs://node01:9000/flink/ha/
high-availability.zookeeper.quorum: node01:2181,node02:2181,node03:2181
```

HA集群测试

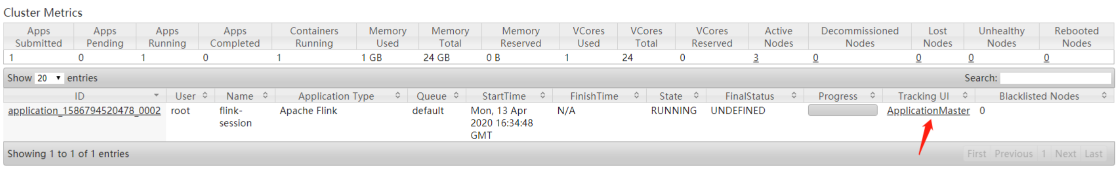
两种模式都可以测试，因为不管哪种模式都会启动yarn-session

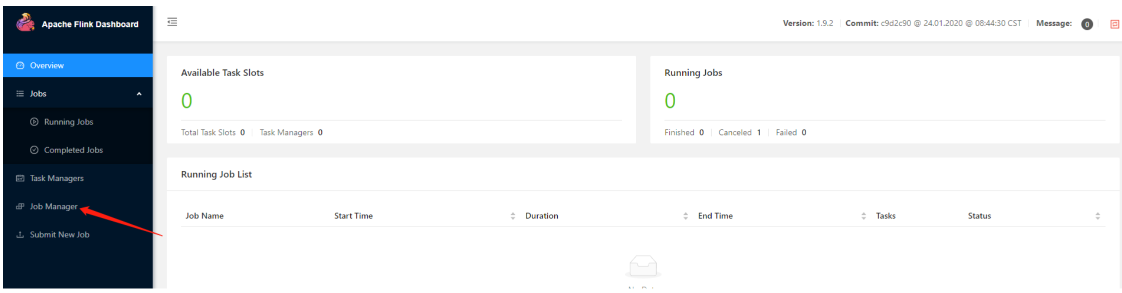
yarn-session模式测试

- 启动yarn-session

```
yarn-session.sh  -n 3 -s 3 -nm flink-session  -d
```

- 通过yarn web ui 找到ApplicationMaster，发现此时的JobManager是在node02启动，现在kill掉JobManager进程 kill -9 进程号





high-availability	zookeeper
high-availability.cluster-id	application_1586794520478_0002
high-availability.storageDir	hdfs://node01:9000/flink/ha/
high-availability.zookeeper.quorum	node01:2181,node02:2181,node03:2181
internal.cluster.execution-mode	NORMAL
internal.io.tmpdirs.use-local-default	true
io.tmp.dirs	/var/msb/hadoop/cluster/nm-local-dir/usercache/root/appcache/application_1586794520478_0002
jobmanager.execution.failover-strategy	region
jobmanager.heap.size	1024m
jobmanager.rpc.address	node02
jobmanager.rpc.port	50132
parallelism.default	1
rest.address	node02
taskmanager.heap.size	1024m

- 再次查看 发现jobManager切换到node03

io.tmp.dirs	/var/msb/hadoop/cluster/nm-local-dir/usercache/root/appcache/application_1586794520478_0002
jobmanager.execution.failover-strategy	region
jobmanager.heap.size	1024m
jobmanager.rpc.address	node03
jobmanager.rpc.port	60599
parallelism.default	1
rest.address	node03

- 查看node03日志

Showing 4096 bytes. Click here for full log

```
:29,655 WARN  akka.remote.ReliableDeliverySupervisor                       - Association with remote system [akka.tcp://flink@node02:50132] has failed, address is now gated for [50] ms. Reason: [Association failed with [akka.tcp://flink@node02:50132]] Caused by: [java.net.ConnectExce
2020-04-08 22:21:29,752 WARN  akka.remote.transport.netty.NettyTransport                   - Remote connection to [null] failed with java.net.ConnectException: Connection refused: node02/192.168.150.112:50132
2020-04-08 22:21:29,754 WARN  akka.remote.ReliableDeliverySupervisor                       - Association with remote system [akka.tcp://flink@node02:50132] has failed, address is now gated for [50] ms. Reason: [Association failed with [akka.tcp://flink@node02:50132]] Caused by: [java
2020-04-08 22:21:29,836 WARN  akka.remote.transport.netty.NettyTransport                   - Remote connection to [null] failed with java.net.ConnectException: Connection refused: node02/192.168.150.112:50132
2020-04-08 22:21:29,840 WARN  akka.remote.ReliableDeliverySupervisor                       - Association with remote system [akka.tcp://flink@node02:50132] has failed, address is now gated for [50] ms. Reason: [Association failed with [akka.tcp://flink@node02:50132]] Caused by: [java
2020-04-08 22:21:29,916 WARN  akka.remote.transport.netty.NettyTransport                   - Remote connection to [null] failed with java.net.ConnectException: Connection refused: node02/192.168.150.112:50132
2020-04-08 22:21:29,917 WARN  akka.remote.ReliableDeliverySupervisor                       - Association with remote system [akka.tcp://flink@node02:50132] has failed, address is now gated for [50] ms. Reason: [Association failed with [akka.tcp://flink@node02:50132]] Caused by: [java
2020-04-08 22:21:29,984 WARN  akka.remote.transport.netty.NettyTransport                   - Remote connection to [null] failed with java.net.ConnectException: Connection refused: node02/192.168.150.112:50132
2020-04-08 22:21:29,987 WARN  akka.remote.ReliableDeliverySupervisor                       - Association with remote system [akka.tcp://flink@node02:50132] has failed, address is now gated for [50] ms. Reason: [Association failed with [akka.tcp://flink@node02:50132]] Caused by: [java
2020-04-08 22:21:30,123 WARN  akka.remote.transport.netty.NettyTransport                   - Remote connection to [null] failed with java.net.ConnectException: Connection refused: node02/192.168.150.112:50132
2020-04-08 22:21:30,125 WARN  akka.remote.ReliableDeliverySupervisor                       - Association with remote system [akka.tcp://flink@node02:50132] has failed, address is now gated for [50] ms. Reason: [Association failed with [akka.tcp://flink@node02:50132]] Caused by: [java
2020-04-08 22:21:36,030 INFO  org.apache.flink.runtime.dispatcher.DispatcherRestEndpoint    - http://node03:41463 was granted leadership with leaderSessionID=9dbf4fc9-d01a-43bc-8d22-bbac6a0c3a65
2020-04-08 22:21:36,041 INFO  org.apache.flink.runtime.dispatcher.StandaloneDispatcher      - Dispatcher akka.tcp://flink@node03:60599/user/dispatcher was granted leadership with fencing token b486e60d-6968-42a4-82a9-b9a109c9c6eb
2020-04-08 22:21:36,043 INFO  org.apache.flink.runtime.dispatcher.StandaloneDispatcher      - Recovering all persisted jobs.
2020-04-08 22:21:36,044 INFO  org.apache.flink.yarn.YarnResourceManager                    - ResourceManager akka.tcp://flink@node03:60599/user/resourcemanager was granted leadership with fencing token 94c94c3d68ed799374303fad7447418b
2020-04-08 22:21:36,048 INFO  org.apache.flink.runtime.resourcemanager.slotmanager.SlotManagerImpl  - Starting the SlotManager.
```

```
2020-04-08 22:21:36,044 INFO  org.apache.flink.yarn.YarnResourceManager
                - ResourceManager
akka.tcp://flink@node03:60599/user/resourcemanager was granted leadership
with fencing token 94c94c3d68ed799374303fad7447418b
```

取消job 开始Run a Flink job on YARN模式测试

flink list

flink canel id

Run a Flink job on YARN模式测试

- 提交job

```
flink run -m yarn-cluster -yn 3 -ys 3 -ynm flink-job -c
com.msb.stream.WordCount ~/StudyFlink-1.0-SNAPSHOT.jar
```

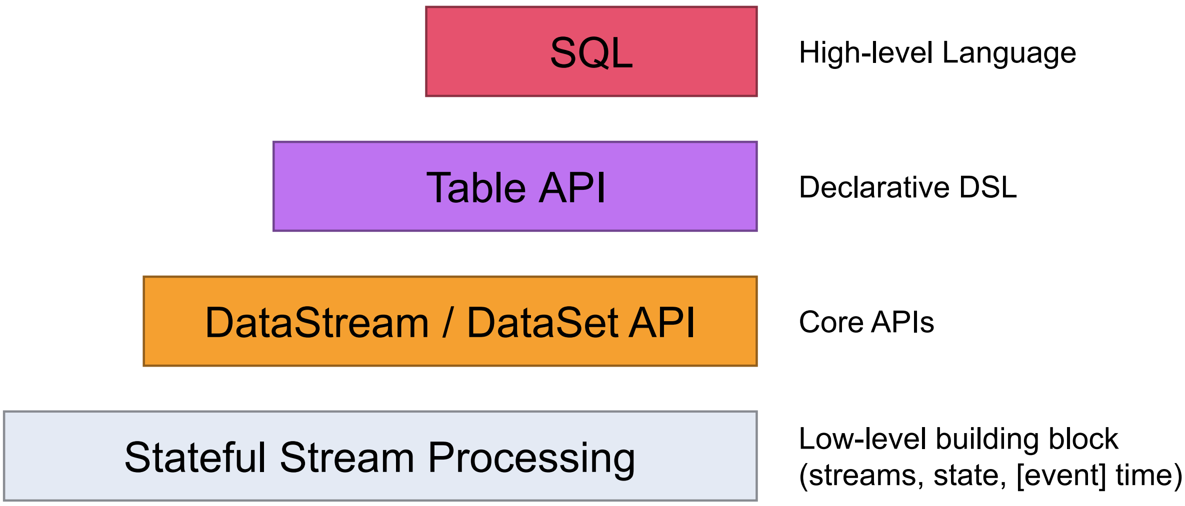
- 停掉JobManager 观察
- 测试完毕，取消job

```
yarn application -kill applicationId
```

Flink API详解&实操

Flink API介绍

Flink提供了不同的抽象级别以开发流式或者批处理应用程序



- **Stateful Stream Processing** 最低级的抽象接口是状态化的数据流接口（stateful streaming）。这个接口是通过 ProcessFunction 集成到 DataStream API 中的。该接口允许用户自由的处理来自一个或多个流中的事件，并使用一致的容错状态。另外，用户也可以通过注册 event time 和 processing time 处理回调函数的方法来实现复杂的计算
- **DataStream/DataSet API** DataStream / DataSet API 是 Flink 提供的核心 API， DataSet 处理有界的数据集，DataStream 处理有界或者无界的数据流。用户可以通过各种方法（map / flatmap / window / keyby / sum / max / min / avg / join 等）将数据进行转换 / 计算
- **Table API** Table API 提供了例如 select、project、join、group-by、aggregate 等操作，使用起来却更加简洁,可以在表与 DataStream/DataSet 之间无缝切换，也允许程序将 Table API 与 DataStream 以及 DataSet 混合使用
- **SQL** Flink 提供的最高层级的抽象是 SQL 。这一层抽象在语法与表达能力上与 Table API 类似。SQL 抽象与 Table API 交互密切，同时 SQL 查询可以直接在 Table API 定义的表上执行

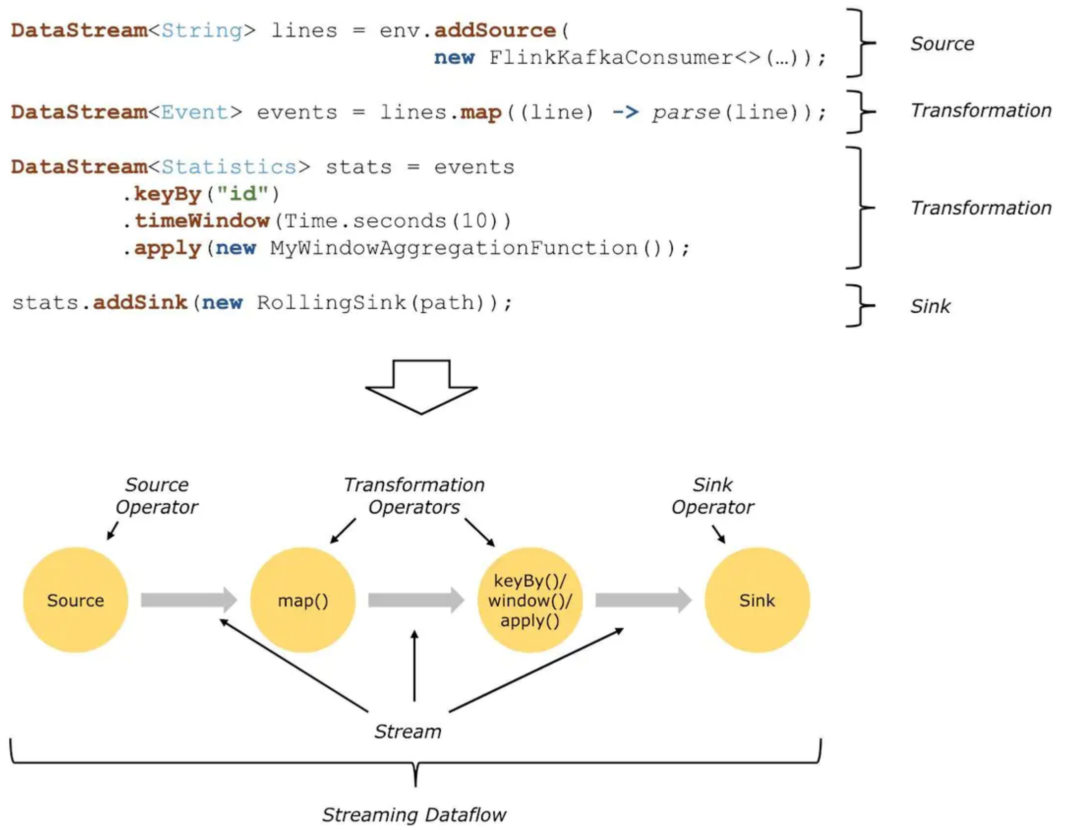
Dataflows数据流图

在Flink的世界观中，一切都是数据流，所以对于批计算来说，那只是流计算的一个特例而已

Flink Dataflows是由三部分组成，分别是：source、transformation、sink结束

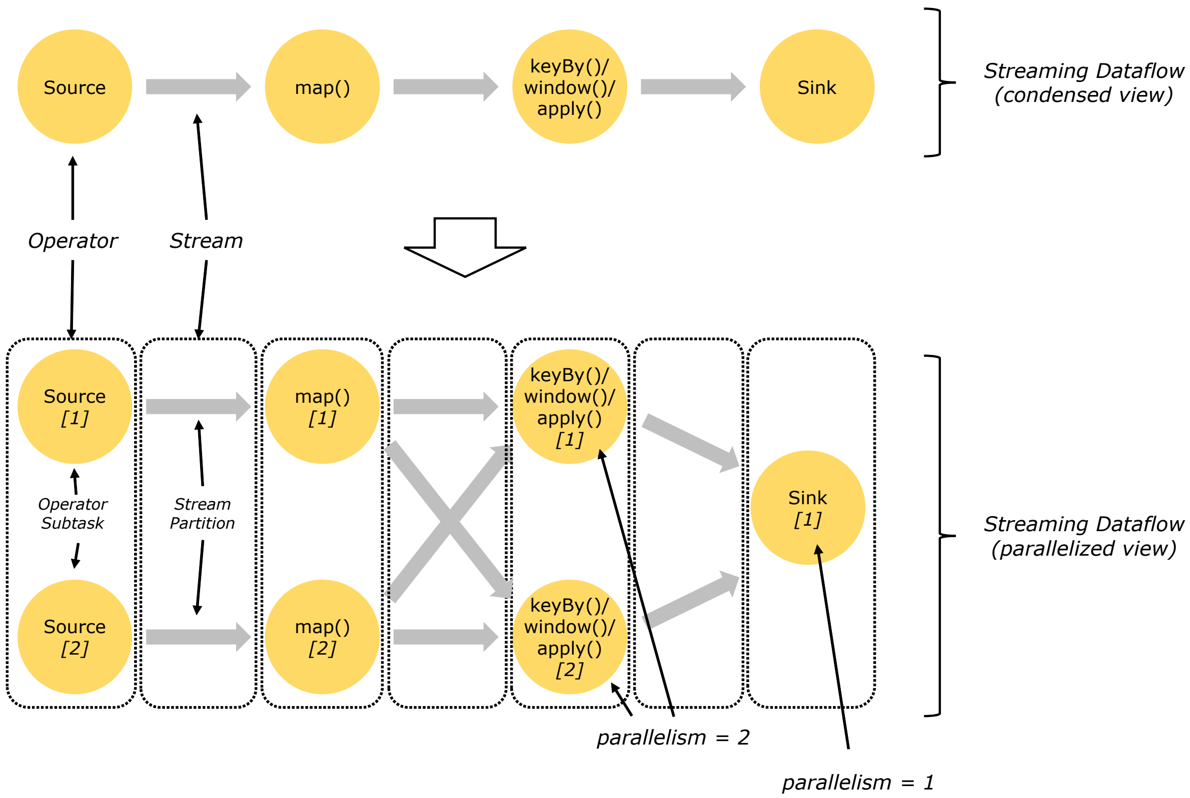
source数据源会源源不断的产生数据，transformation将产生的数据进行各种业务逻辑的数据处理，最终由sink输出到外部（console、kafka、redis、DB......）

基于Flink开发的程序都能够映射成一个Dataflows



当source数据源的数量比较大或计算逻辑相对比较复杂的情况下，需要提高并行度来处理数据，采用并行数据流

通过设置不同算子的并行度 source并行度设置为2 map也是2.... 代表会启动多个并行的线程来处理数据



配置开发环境

每个 Flink 应用都需要依赖一组 Flink 类库。Flink 应用至少需要依赖 Flink APIs。许多应用还会额外依赖连接器类库(比如 Kafka、Cassandra 等)。 当用户运行 Flink 应用时(无论是在 IDEA 环境下进行测试，还是部署在分布式环境下)，运行时类库都必须可用

开发工具：IntelliJ IDEA

配置开发Maven依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.11</artifactId>
  <version>1.10.0</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.11</artifactId>
  <version>1.10.0</version>
</dependency>
```

注意点：

- 如果要将程序打包提交到集群运行，打包的时候不需要包含这些依赖，因为集群环境已经包含了这些依赖，此时依赖的作用域应该设置为provided provided
- Flink 应用在 IntelliJ IDEA 中运行，这些 Flink 核心依赖的作用域需要设置为 *compile* 而不是 *provided* 。 否则 IntelliJ 不会添加这些依赖到 classpath，会导致应用运行时抛出 `NoClassDefFountError` 异常

添加打包插件：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.1.1</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <excludes>

<exclude>com.google.code.findbugs:jsr305</exclude>
                                <exclude>org.slf4j:*</exclude>
                                <exclude>log4j:*</exclude>
                            </excludes>
                        </artifactSet>
                        <filters>
                            <filter>
                                <!--不要拷贝 META-INF 目录下的签名，
                                否则会引起 SecurityExceptions 。 -->
                                <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>my.programs.main.clazz</mainClass>
                            </transformer>
                        </transformers>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

WordCount流批计算程序

批计算：统计HDFS文件单词出现的次数

读取HDFS数据需要添加Hadoop依赖

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.6.5</version>
</dependency>
```

WordCount代码：

```
  val env = ExecutionEnvironment.getExecutionEnvironment
    val initDS: DataSet[String] =
env.readTextFile("hdfs://node01:9000/flink/data/wc")
    val restDS: AggregateDataSet[(String, Int)] = initDS.flatMap(_.split("
")).map((_,1)).groupBy(0).sum(1)
    restDS.print()
```

流计算：统计数据流中，单词出现的次数

```
//准备环境
    /**
      * createLocalEnvironment 创建一个本地执行的环境  local
      * createLocalEnvironmentWithWebUI 创建一个本地执行的环境  同时还开启Web UI的查看
端口  8081
      * getExecutionEnvironment 根据你执行的环境创建上下文，比如local  cluster
      */
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)
    /**
      * DataStream：一组相同类型的元素 组成的数据流
      */
    val initStream:DataStream[String] = env.socketTextStream("node01",8888)
```

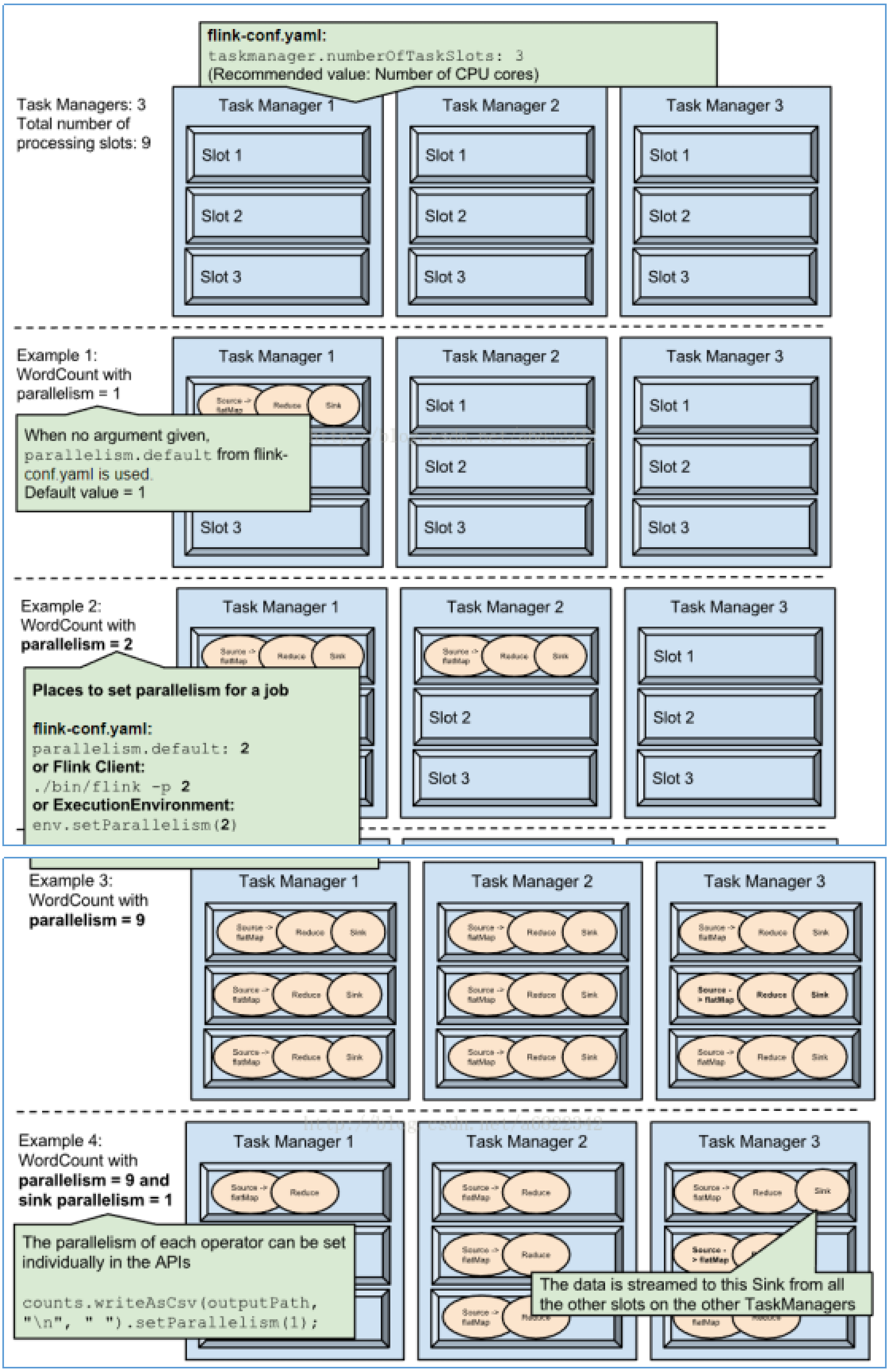
```
    val wordStream = initStream.flatMap(_.split(" "))
    val pairStream = wordStream.map((_,1))
    val keyByStream = pairStream.keyBy(0)
    val restStream = keyByStream.sum(1)
    restStream.print()

    /**
      * 6> (msb,1)
      * 1> (,,1)
      * 3> (hello,1)
      * 3> (hello,2)
      * 6> (msb,2)
      * 默认就是有状态的计算
      * 6>  代表是哪一个线程处理的
      *
      * 相同的数据一定是由某一个thread处理
      **/
    //启动Flink 任务
    env.execute("first flink job")
```

WordCount Dataflows 算子链

为了更高效地分布式执行，Flink会尽可能地将operator的subtask链接（chain）在一起形成task。每个task在一个线程中执行。将operators链接成task是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量

Flink任务调度规则

- 不同Task下的subtask分到同一个TaskSlot，提高数据传输效率
- 相同Task下的subtask不会分到同一个TaskSlot，充分利用集群资源



Flink并行度设置方式

1. 在算子上设置

```
val wordStream = initStream.flatMap(_.split(" ")).setParallelism(2)
```

2. 在上下文环境中设置

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1)
```

3. client提交Job时设置

```
flink run -c com.msb.stream.WordCount -p 3 StudyFlink-1.0-SNAPSHOT.jar
```

4. 在flink-conf.yaml配置文件中设置

```
parallelism.default: 1
```

这四种设置并行度的方式，优先级依次递减

Dataflows DataSource数据源

Flink内嵌支持的数据源非常多，比如HDFS、Socket、Kafka、Collections Flink也提供了addSource方式，可以自定义数据源，本小节将讲解Flink所有内嵌数据源及自定义数据源的原理及API

File Source

- 通过读取本地、HDFS文件创建一个数据源

如果读取的是HDFS上的文件，那么需要导入Hadoop依赖

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.6.5</version>
</dependency>
```

代码：

```
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
//在算子转换的时候，会将数据转换成Flink内置的数据类型，所以需要将隐式转换导入进来，才能自动进行类型转换
import org.apache.flink.streaming.api.scala._

object FileSource {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val textStream = env.readTextFile("hdfs://node01:9000/flink/data/wc")
    textStream.flatMap(_.split(" ")).map((_,1)).keyBy(0).sum(1).print()
    //读完就停止
    env.execute()
  }
}
```

- 每隔10s中读取HDFS指定目录下的新增文件内容，并且进行WordCount

  业务场景：在企业中一般都会做实时的ETL，当Flume采集来新的数据，那么基于Flink实时做ETL入仓

```
import org.apache.flink.api.java.io.TextInputFormat
import org.apache.flink.core.fs.Path
import org.apache.flink.streaming.api.functions.source.FileProcessingMode
```

```scala
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
//在算子转换的时候，会将数据转换成Flink内置的数据类型，所以需要将隐式转换导入进来，才能自动进行类型转换
import org.apache.flink.streaming.api.scala._

object FileSource {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //读取hdfs文件
    val filePath = "hdfs://node01:9000/flink/data/"
    val textInputFormat = new TextInputFormat(new Path(filePath))
    //每隔10s中读取 hdfs上新增文件内容
    val textStream =
env.readFile(textInputFormat,filePath,FileProcessingMode.PROCESS_CONTINUOUSLY,10
)
//    val textStream = env.readTextFile("hdfs://node01:9000/flink/data/wc")
    textStream.flatMap(_.split(" ")).map((_,1)).keyBy(0).sum(1).print()
    env.execute()
  }
}
```

readTextFile底层调用的就是readFile方法，readFile是一个更加底层的方式，使用起来会更加的灵活

Collection Source

基于本地集合的数据源，一般用于测试场景，没有太大意义

```scala
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._

object CollectionSource {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.fromCollection(List("hello flink msb","hello msb msb"))
    stream.flatMap(_.split(" ")).map((_,1)).keyBy(0).sum(1).print()
    env.execute()
  }
}
```

Socket Source

接受Socket Server中的数据，已经讲过

```scala
val initStream:DataStream[String] = env.socketTextStream("node01",8888)
```

Kafka Source

Flink接受Kafka中的数据，首先先配置flink与kafka的连接器依赖

官网地址：https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/connectors/kafka.html

maven依赖

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.11</artifactId>
  <version>1.9.2</version>
</dependency>
```

代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
    val prop = new Properties()
    prop.setProperty("bootstrap.servers","node01:9092,node02:9092,node03:9092")
    prop.setProperty("group.id","flink-kafka-id001")
    prop.setProperty("key.deserializer",classOf[StringDeserializer].getName)
    prop.setProperty("value.deserializer",classOf[StringDeserializer].getName)
    /**
      * earliest:从头开始消费，旧数据会频繁消费
      * latest:从最近的数据开始消费，不再消费旧数据
      */
    prop.setProperty("auto.offset.reset","latest")

    val kafkaStream = env.addSource(new FlinkKafkaConsumer[(String, String)]
("flink-kafka", new KafkaDeserializationSchema[(String, String)] {
      override def isEndOfStream(t: (String, String)): Boolean = false

      override def deserialize(consumerRecord: ConsumerRecord[Array[Byte],
Array[Byte]]): (String, String) = {
        val key = new String(consumerRecord.key(), "UTF-8")
        val value = new String(consumerRecord.value(), "UTF-8")
        (key, value)
      }
      //指定返回数据类型
      override def getProducedType: TypeInformation[(String, String)] =
        createTuple2TypeInformation(createTypeInformation[String],
createTypeInformation[String])
    }, prop))
    kafkaStream.print()
    env.execute()
```

kafka命令消费key value值

kafka-console-consumer.sh --zookeeper node01:2181 --topic flink-kafka --property print.key=true

默认只是消费value值

KafkaDeserializationSchema：读取kafka中key、value

SimpleStringSchema：读取kafka中value

Custom Source

Sources are where your program reads its input from. You can attach a source to your program by using `StreamExecutionEnvironment.addSource(sourceFunction)`. Flink comes with a number of pre-implemented source functions, but you can always write your own custom sources by implementing the `SourceFunction` for non-parallel sources, or by implementing the `ParallelSourceFunction` interface or extending the `RichParallelSourceFunction` for parallel sources.

- 基于SourceFunction接口实现单并行度数据源

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
    //source的并行度为1 单并行度source源
    val stream = env.addSource(new SourceFunction[String] {
      var flag = true
      override def run(ctx: SourceFunction.SourceContext[String]): Unit = {
        val random = new Random()
        while (flag) {
          ctx.collect("hello" + random.nextInt(1000))
          Thread.sleep(200)
        }
      }
      //停止产生数据
      override def cancel(): Unit = flag = false
    })
    stream.print()
    env.execute()
```

基于ParallelSourceFunction接口实现多并行度数据源

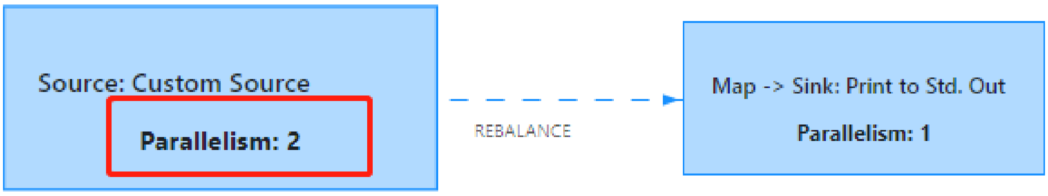
```
  public interface ParallelSourceFunction<OUT> extends SourceFunction<OUT> {}
```

```
  public abstract class RichParallelSourceFunction<OUT> extends
AbstractRichFunction
        implements ParallelSourceFunction<OUT> {
    private static final long serialVersionUID = 1L;
  }
```

实现ParallelSourceFunction接口=继承RichParallelSourceFunction

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sourceStream = env.addSource(new ParallelSourceFunction[String] {
      var flag = true

      override def run(ctx: SourceFunction.SourceContext[String]): Unit = {
        val random = new Random()
        while (flag) {
          ctx.collect("hello" + random.nextInt(1000))
          Thread.sleep(500)
        }
      }

      override def cancel(): Unit = {
        flag = false
      }
    }).setParallelism(2)
```

数据源可以设置为多并行度



Dataflows Transformations

Transformations算子可以将一个或者多个算子转换成一个新的数据流，使用Transformations算子组合可以进行复杂的业务处理

Map

DataStream → DataStream

遍历数据流中的每一个元素，产生一个新的元素

FlatMap

DataStream → DataStream

遍历数据流中的每一个元素，产生N个元素 N=0，1，2,......

Filter

DataStream → DataStream

过滤算子，根据数据流的元素计算出一个boolean类型的值，true代表保留，false代表过滤掉

KeyBy

DataStream → KeyedStream

根据数据流中指定的字段来分区，相同指定字段值的数据一定是在同一个分区中，内部分区使用的是HashPartitioner

指定分区字段的方式有三种：

```
1、根据索引号指定
2、通过匿名函数来指定
3、通过实现KeySelector接口  指定分区字段

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.generateSequence(1, 100)
    stream
      .map(x => (x % 3, 1))
      //根据索引号来指定分区字段
      //      .keyBy(0)
      //通过传入匿名函数 指定分区字段
      //      .keyBy(x=>x._1)
      //通过实现KeySelector接口  指定分区字段
      .keyBy(new KeySelector[(Long, Int), Long] {
      override def getKey(value: (Long, Int)): Long = value._1
    })
```

```
        .sum(1)
        .print()
      env.execute()
```

Reduce

KeyedStream：根据key分组 → DataStream

注意，reduce是基于分区后的流对象进行聚合，也就是说，DataStream类型的对象无法调用reduce方法

```
.reduce((v1,v2) => (v1._1,v1._2 + v2._2))
```

demo01：读取kafka数据，实时统计各个卡口下的车流量

- 实现kafka生产者，读取卡口数据并且往kafka中生产数据

```
    val prop = new Properties()
    prop.setProperty("bootstrap.servers", "node01:9092,node02:9092,node03:9092")
    prop.setProperty("key.serializer", classOf[StringSerializer].getName)
    prop.setProperty("value.serializer", classOf[StringSerializer].getName)

    val producer = new KafkaProducer[String, String](prop)

    val iterator = Source.fromFile("data/carFlow_all_column_test.txt", "UTF-
8").getLines()
    for (i <- 1 to 100) {
      for (line <- iterator) {
        //将需要的字段值 生产到kafka集群  car_id monitor_id event-time speed
        //车牌号 卡口号 车辆通过时间 通过速度
        val splits = line.split(",")
        val monitorID = splits(0).replace("'","")
        val car_id = splits(2).replace("'","")
        val eventTime = splits(4).replace("'","")
        val speed = splits(6).replace("'","")
        if (!"00000000".equals(car_id)) {
          val event = new StringBuilder
          event.append(monitorID + "\t").append(car_id+"\t").append(eventTime +
"\t").append(speed)
          producer.send(new ProducerRecord[String, String]("flink-kafka",
event.toString()))
        }

        Thread.sleep(500)
      }
    }
```

- 实现代码

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
    val props = new Properties()
    props.setProperty("bootstrap.servers","node01:9092,node02:9092,node03:9092")
    props.setProperty("key.deserializer",classOf[StringDeserializer].getName)
    props.setProperty("value.deserializer",classOf[StringDeserializer].getName)
    props.setProperty("group.id","flink001")
    props.getProperty("auto.offset.reset","latest")
```

```
    val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new
SimpleStringSchema(),props))
    stream.map(data => {
      val splits = data.split("\t")
      val carFlow = CarFlow(splits(0),splits(1),splits(2),splits(3).toDouble)
      (carFlow,1)
    }).keyBy(_._1.monitorId)
        .sum(1)
        .print()
    env.execute()
```

Aggregations

KeyedStream → DataStream

Aggregations代表的是一类聚合算子，具体算子如下：

```
keyedStream.sum(0)
keyedStream.sum("key")
keyedStream.min(0)
keyedStream.min("key")
keyedStream.max(0)
keyedStream.max("key")
keyedStream.minBy(0)
keyedStream.minBy("key")
keyedStream.maxBy(0)
keyedStream.maxBy("key")
```

demo02：实时统计各个卡口最先通过的汽车的信息

```
val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new
SimpleStringSchema(),props))
    stream.map(data => {
      val splits = data.split("\t")
      val carFlow = CarFlow(splits(0),splits(1),splits(2),splits(3).toDouble)
      val eventTime = carFlow.eventTime
      val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
      val date = format.parse(eventTime)
      (carFlow,date.getTime)
    }).keyBy(_._1.monitorId)
        .min(1)
        .map(_._1)
        .print()
    env.execute()
```

Union 真合并

DataStream* → DataStream

Union of two or more data streams creating a new stream containing all the elements from all the streams

合并两个或者更多的数据流产生一个新的数据流，这个新的数据流中包含了所合并的数据流的元素

注意：需要保证数据流中元素类型一致

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
    val ds1 = env.fromCollection(List(("a",1),("b",2),("c",3)))
    val ds2 = env.fromCollection(List(("d",4),("e",5),("f",6)))
    val ds3 = env.fromCollection(List(("g",7),("h",8)))
//    val ds3 = env.fromCollection(List((1,1),(2,2)))
    val unionStream = ds1.union(ds2,ds3)
    unionStream.print()
    env.execute()
```

Connect 假合并

DataStream,DataStream → ConnectedStreams

合并两个数据流并且保留两个数据流的数据类型，能够共享两个流的状态

```
val ds1 = env.socketTextStream("node01", 8888)
val ds2 = env.socketTextStream("node01", 9999)
val wcStream1 = ds1.flatMap(_.split(" ")).map((_, 1)).keyBy(0).sum(1)
val wcStream2 = ds2.flatMap(_.split(" ")).map((_, 1)).keyBy(0).sum(1)
val restStream: ConnectedStreams[(String, Int), (String, Int)] =
wcStream2.connect(wcStream1)
```

CoMap, CoFlatMap

ConnectedStreams → DataStream

CoMap, CoFlatMap并不是具体算子名字，而是一类操作名称

凡是基于ConnectedStreams数据流做map遍历，这类操作叫做CoMap

凡是基于ConnectedStreams数据流做flatMap遍历，这类操作叫做CoFlatMap

CoMap第一种实现方式：

```
restStream.map(new CoMapFunction[(String,Int),(String,Int),(String,Int)] {
      //对第一个数据流做计算
      override def map1(value: (String, Int)): (String, Int) = {
        (value._1+":first",value._2+100)
      }
      //对第二个数据流做计算
      override def map2(value: (String, Int)): (String, Int) = {
        (value._1+":second",value._2*100)
      }
    }).print()
```

CoMap第二种实现方式：

```
restStream.map(
      //对第一个数据流做计算
      x=>{(x._1+":first",x._2+100)}
      //对第二个数据流做计算
      ,y=>{(y._1+":second",y._2*100)}
    ).print()
```

CoFlatMap第一种实现方式：

```
ds1.connect(ds2).flatMap((x,c:Collector[String])=>{
      //对第一个数据流做计算
      x.split(" ").foreach(w=>{
        c.collect(w)
      })

    }
      //对第二个数据流做计算
      ,(y,c:Collector[String])=>{
      y.split(" ").foreach(d=>{
        c.collect(d)
      })
    }).print
```

CoFlatMap第二种实现方式：

```
 ds1.connect(ds2).flatMap(
      //对第一个数据流做计算
      x=>{
      x.split(" ")
    }
      //对第二个数据流做计算
      ,y=>{
        y.split(" ")
      }).print()
```

CoFlatMap第三种实现方式：

```
ds1.connect(ds2).flatMap(new CoFlatMapFunction[String,String,(String,Int)] {
    //对第一个数据流做计算
    override def flatMap1(value: String, out: Collector[(String, Int)]): Unit =
{
        val words = value.split(" ")
        words.foreach(x=>{
          out.collect((x,1))
        })
      }

    //对第二个数据流做计算
    override def flatMap2(value: String, out: Collector[(String, Int)]): Unit =
{
        val words = value.split(" ")
        words.foreach(x=>{
          out.collect((x,1))
        })
      }
    }).print()
```

demo03：现有一个配置文件存储车牌号与车主的真实姓名，通过数据流中的车牌号实时匹配出对应的车主姓名（注意：配置文件可能实时改变）

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1)
val filePath = "data/carId2Name"
```

```
val carId2NameStream = env.readFile(new TextInputFormat(new
Path(filePath)),filePath,FileProcessingMode.PROCESS_CONTINUOUSLY,10)
val dataStream = env.socketTextStream("node01",8888)
dataStream.connect(carId2NameStream).map(new CoMapFunction[String,String,String]
{
    private val hashMap = new mutable.HashMap[String,String]()
    override def map1(value: String): String = {
        hashMap.getOrElse(value,"not found name")
    }

    override def map2(value: String): String = {
        val splits = value.split(" ")
        hashMap.put(splits(0),splits(1))
        value + "加载完毕..."
    }
}).print()
env.execute()
此demo仅限深度理解connect算子和CoMap操作，后期还需使用广播流优化
```

Split

DataStream → SplitStream

根据条件将一个流分成两个或者更多的流

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,100)
val splitStream = stream.split(
    d => {
        d % 2 match {
            case 0 => List("even")
            case 1 => List("odd")
        }
    }
)
splitStream.select("even").print()
env.execute()
```

```
@deprecated Please use side output instead
```

Select

SplitStream → DataStream

从SplitStream中选择一个或者多个数据流

```
splitStream.select("even").print()
```

side output侧输出流

流计算过程，可能遇到根据不同的条件来分隔数据流。filter分割造成不必要的数据复制

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.socketTextStream("node01",8888)
    val gtTag = new OutputTag[String]("gt")
    val processStream = stream.process(new ProcessFunction[String, String] {
```

```
      override def processElement(value: String, ctx: ProcessFunction[String,
String]#Context, out: Collector[String]): Unit = {
        try {
          val longVar = value.toLong
          if (longVar > 100) {
            out.collect(value)
          } else {
            ctx.output(gtTag, value)
          }
        } catch {
          case e => e.getMessage
            ctx.output(gtTag, value)
        }
      }
    })
    val sideStream = processStream.getSideOutput(gtTag)
    sideStream.print("sideStream")
    processStream.print("mainStream")
    env.execute()
```

Iterate（比较重要）

DataStream → IterativeStream → DataStream

Iterate算子提供了对数据流迭代的支持

迭代由两部分组成：迭代体、终止迭代条件

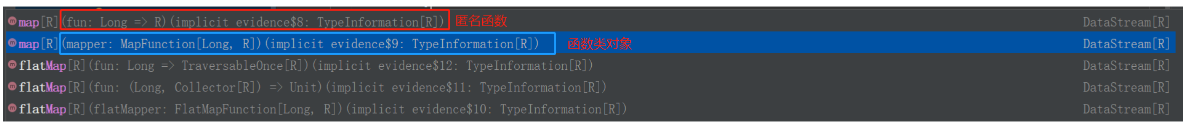
不满足终止迭代条件的数据流会返回到stream流中，进行下一次迭代

满足终止迭代条件的数据流继续往下游发送

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val initStream = env.socketTextStream("node01",8888)
val stream = initStream.map(_.toLong)
stream.iterate {
    iteration => {
        //定义迭代逻辑
        val iterationBody = iteration.map ( x => {
            println(x)
            if(x > 0) x - 1
            else x
        } )
        //> 0  大于0的值继续返回到stream流中,当 <= 0 继续往下游发送
        (iterationBody.filter(_ > 0), iterationBody.filter(_ <= 0))
    }
}.print()
env.execute()
```

函数类和富函数类

在使用Flink算子的时候，可以通过传入匿名函数和函数类对象 例如：



函数类分为：普通函数类、富函数类**（自行划分）**

富函数类相比于普通的函数，可以获取运行环境的上下文（Context），拥有一些生命周期方法，管理状态，可以实现更加复杂的功能

普通函数类	富函数类
MapFunction	RichMapFunction
FlatMapFunction	RichFlatMapFunction
FilterFunction	RichFilterFunction
……	……

- 使用普通函数类过滤掉车速高于100的车辆信息

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.readTextFile("./data/carFlow_all_column_test.txt")
    stream.filter(new FilterFunction[String] {
      override def filter(value: String): Boolean = {
        if (value != null && !"".equals(value)) {
          val speed = value.split(",")(6).replace("'", "").toLong
          if (speed > 100)
            false
          else
            true
        }else
          false
      }
    }).print()
    env.execute()
```

- 使用富函数类，将车牌号转化成车主真实姓名，映射表存储在Redis中

```
@Public
public abstract class RichMapFunction<IN, OUT> extends AbstractRichFunction
implements MapFunction<IN, OUT> {

    private static final long serialVersionUID = 1L;

    @Override
    public abstract OUT map(IN value) throws Exception;
}

public abstract class AbstractRichFunction implements RichFunction, Serializable
{
    @Override
    public void open(Configuration parameters) throws Exception {}

    @Override
    public void close() throws Exception {}
}
```

abstract class RichMapFunction实现MapFunction接口

map函数是抽象方法，需要实现

添加redis依赖

wordcount数据写入到redis

```
<dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>${redis.version}</version>
</dependency>
```

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.socketTextStream("node01", 8888)
    stream.map(new RichMapFunction[String, String] {

      private var jedis: Jedis = _

      //初始化函数  在每一个thread启动的时候（处理元素的时候，会调用一次）
      //在open中可以创建连接redis的连接
      override def open(parameters: Configuration): Unit = {
        //getRuntimeContext可以获取flink运行的上下文环境  AbstractRichFunction抽象类
提供的
        val taskName = getRuntimeContext.getTaskName
        val subtasks = getRuntimeContext.getTaskNameWithSubtasks
        println("=========open======"+"taskName:" + taskName +
"\tsubtasks:"+subtasks)
        jedis = new Jedis("node01", 6379)
        jedis.select(3)
      }

      //每处理一个元素，就会调用一次
      override def map(value: String): String = {
        val name = jedis.get(value)
        if(name == null){
          "not found name"
        }else
          name
      }

      //元素处理完毕后，会调用close方法
      //关闭redis连接
      override def close(): Unit = {
        jedis.close()
      }
    }).setParallelism(2).print()

    env.execute()
```

底层API(ProcessFunctionAPI)

属于低层次的API，我们前面讲的map、filter、flatMap等算子都是基于这层高层封装出来的

越低层次的API，功能越强大，用户能够获取的信息越多，比如可以拿到元素状态信息、事件时间、设置定时器等

- 监控每辆汽车，车速超过100迈，5s钟后发出超速的警告通知

```
object MonitorOverSpeed02 {
  case class CarInfo(carId:String,speed:Long)
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.socketTextStream("node01",8888)
    stream.map(data => {
      val splits = data.split(" ")
      val carId = splits(0)
      val speed = splits(1).toLong
      CarInfo(carId,speed)
    }).keyBy(_.carId)
      //KeyedStream调用process需要传入KeyedProcessFunction
      //DataStream调用process需要传入ProcessFunction
      .process(new KeyedProcessFunction[String,CarInfo,String] {

      override def processElement(value: CarInfo, ctx:
KeyedProcessFunction[String, CarInfo, String]#Context, out:
Collector[String]): Unit = {
        val currentTime = ctx.timerService().currentProcessingTime()
        if(value.speed > 100 ){
          val timerTime = currentTime + 2 * 1000
          ctx.timerService().registerProcessingTimeTimer(timerTime)
        }
      }

      override def onTimer(timestamp: Long, ctx:
KeyedProcessFunction[String, CarInfo, String]#OnTimerContext, out:
Collector[String]): Unit = {
        var warnMsg = "warn... time:" + timestamp + "  carID:" +
ctx.getCurrentKey
        out.collect(warnMsg)
      }
    }).print()

    env.execute()
  }
}
```

总结

Dataflows分区策略

shuffle

场景：增大分区、提高并行度，解决数据倾斜

DataStream → DataStream

分区元素随机均匀分发到下游分区，网络开销比较大

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10).setParallelism(1)
println(stream.getParallelism)
stream.shuffle.print()
env.execute()
```

console result: 上游数据比较随意的分发到下游

```
2> 1
1> 4
7> 10
4> 6
6> 3
5> 7
8> 2
1> 5
1> 8
1> 9
```

rebalance

场景：增大分区、提高并行度，解决数据倾斜

DataStream → DataStream

轮询分区元素，均匀的将元素分发到下游分区，下游每个分区的数据比较均匀，在发生数据倾斜时非常有用，网络开销比较大

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(3)
val stream = env.generateSequence(1,100)
val shuffleStream = stream.rebalance
shuffleStream.print()
env.execute()
```

console result:上游数据比较均匀的分发到下游

```
8> 6
3> 1
5> 3
7> 5
1> 7
2> 8
6> 4
4> 2
3> 9
4> 10
```

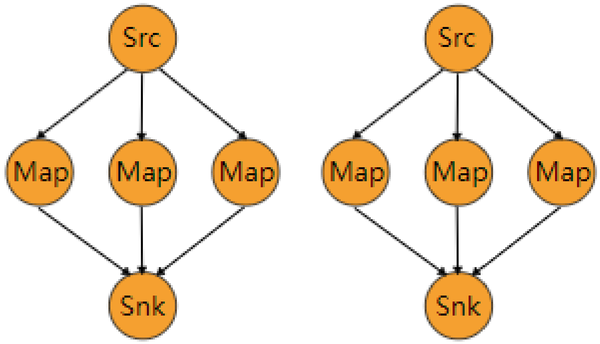
rescale

场景：减少分区 防止发生大量的网络传输 不会发生全量的重分区

DataStream → DataStream

通过轮询分区元素，将一个元素集合从上游分区发送给下游分区，发送单位是集合，而不是一个个元素

注意：rescale发生的是本地数据传输，而不需要通过网络传输数据，比如taskmanager的槽数。简单来说，上游的数据只会发送给本TaskManager中的下游




```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10).setParallelism(2)
stream.writeAsText("./data/stream1").setParallelism(2)
stream.rescale.writeAsText("./data/stream2").setParallelism(4)
env.execute()
```

console result：stream1:1内容 分发给stream2:1和stream2:2

stream1:1

```
1
3
5
7
9
```

stream1:2

```
2
4
6
8
10
```

stream2:1

```
1
5
9
```

stream2:2

```
3
7
```

stream2:3

```
2
6
10
```

stream2:4

```
4
8
```

broadcast

场景：需要使用映射表、并且映射表会经常发生变动的场景

DataStream → DataStream

上游中每一个元素内容广播到下游每一个分区中

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10).setParallelism(2)
stream.writeAsText("./data/stream1").setParallelism(2)
stream.broadcast.writeAsText("./data/stream2").setParallelism(4)
env.execute()
```

console result：stream1:1、2内容广播到了下游每个分区中

stream1:1

```
1
3
5
7
9
```

stream1:2

```
2
4
6
8
10
```

stream2:1

```
1
3
5
7
9
2
4
6
8
10
```

global

场景：并行度降为1

DataStream → DataStream

上游分区的数据只分发给下游的第一个分区

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10).setParallelism(2)
stream.writeAsText("./data/stream1").setParallelism(2)
stream.global.writeAsText("./data/stream2").setParallelism(4)
env.execute()
```

console result：stream1:1、2内容只分发给了stream2:1

stream1:1

```
1
3
5
7
9
```

stream1:2

```
2
4
6
8
10
```

stream2:1

```
1
3
5
7
9
2
4
6
8
10
```

forward

场景：一对一的数据分发，map、flatMap、filter 等都是这种分区策略

DataStream → DataStream

上游分区数据分发到下游对应分区中

partition1->partition1

partition2->partition2

注意：必须保证上下游分区数（并行度）一致，不然会有如下异常：

```
Forward partitioning does not allow change of parallelism
* Upstream operation: Source: Sequence Source-1 parallelism: 2,
* downstream operation: Sink: Unnamed-4 parallelism: 4
* stream.forward.writeAsText("./data/stream2").setParallelism(4)
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10).setParallelism(2)
stream.writeAsText("./data/stream1").setParallelism(2)
stream.forward.writeAsText("./data/stream2").setParallelism(2)
env.execute()
```

console result：stream1:1->stream2:1、stream1:2->stream2:2

stream1:1

```
1
3
5
7
9
```

stream1:2

```
2
4
6
8
10
```

stream2:1

```
1
3
5
7
9
```

stream2:2

```
2
4
6
8
10
```

keyBy

场景：与业务场景匹配

DataStream → DataStream

根据上游分区元素的Hash值与下游分区数取模计算出，将当前元素分发到下游哪一个分区

```
MathUtils.murmurHash(keyHash)（每个元素的Hash值） % maxParallelism（下游分区数）
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10).setParallelism(2)
stream.writeAsText("./data/stream1").setParallelism(2)
stream.keyBy(0).writeAsText("./data/stream2").setParallelism(2)
env.execute()
```

console result：根据元素Hash值分发到下游分区中

PartitionCustom

DataStream → DataStream

通过自定义的分区器，来决定元素是如何从上游分区分发到下游分区

```
object ShuffleOperator {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(2)
    val stream = env.generateSequence(1,10).map((_,1))
    stream.writeAsText("./data/stream1")
    stream.partitionCustom(new customPartitioner(),0)
      .writeAsText("./data/stream2").setParallelism(4)
    env.execute()
  }
  class customPartitioner extends Partitioner[Long]{
    override def partition(key: Long, numPartitions: Int): Int = {
      key.toInt % numPartitions
    }
  }
}
```

Dataflows Sink

Flink内置了大量sink，可以将Flink处理后的数据输出到HDFS、kafka、Redis、ES、MySQL等等

工程场景中，会经常消费kafka中数据，处理结果存储到Redis或者MySQL中

Redis Sink

Flink处理的数据可以存储到Redis中，以便实时查询

Flink内嵌连接Redis的连接器，只需要导入连接Redis的依赖就可以

```
        <dependency>
            <groupId>org.apache.bahir</groupId>
            <artifactId>flink-connector-redis_2.11</artifactId>
            <version>1.0</version>
        </dependency>
```

WordCount写入到Redis中，选择的是HSET数据类型

代码如下：

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.socketTextStream("node01",8888)
    val result = stream.flatMap(_.split(" "))
      .map((_, 1))
      .keyBy(0)
      .sum(1)

    //若redis是单机
    val config = new
FlinkJedisPoolConfig.Builder().setDatabase(3).setHost("node01").setPort(6379).bu
ild()
    //如果是 redis集群
    /*val addresses = new util.HashSet[InetSocketAddress]()
    addresses.add(new InetSocketAddress("node01",6379))
    addresses.add(new InetSocketAddress("node01",6379))
   val clusterConfig = new
FlinkJedisClusterConfig.Builder().setNodes(addresses).build()*/

    result.addSink(new RedisSink[(String,Int)](config,new
RedisMapper[(String,Int)] {

      override def getCommandDescription: RedisCommandDescription = {
        new RedisCommandDescription(RedisCommand.HSET,"wc")
      }

      override def getKeyFromData(t: (String, Int))  = {
        t._1
      }

      override def getValueFromData(t: (String, Int))  = {
        t._2 + ""
      }
    }))
    env.execute()
```

Kafka Sink

处理结果写入到kafka topic中，Flink也是默认支持，需要添加连接器依赖，跟读取kafka数据用的连接器依赖相同

之前添加过就不需要再次添加了

```
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_2.11</artifactId>
            <version>${flink-version}</version>
        </dependency>
```

```
import java.lang
import java.util.Properties

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.connectors.kafka.{FlinkKafkaProducer,
KafkaSerializationSchema}
import org.apache.kafka.clients.producer.ProducerRecord
```

```
import org.apache.kafka.common.serialization.StringSerializer

object KafkaSink {
  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val stream = env.socketTextStream("node01",8888)
    val result = stream.flatMap(_.split(" "))
      .map((_, 1))
      .keyBy(0)
      .sum(1)

    val props = new Properties()
    props.setProperty("bootstrap.servers","node01:9092,node02:9092,node03:9092")
//    props.setProperty("key.serializer",classOf[StringSerializer].getName)
//    props.setProperty("value.serializer",classOf[StringSerializer].getName)


    /**
    public FlinkKafkaProducer(
     FlinkKafkaProducer(defaultTopic: String, serializationSchema:
KafkaSerializationSchema[IN], producerConfig: Properties, semantic:
FlinkKafkaProducer.Semantic)
      */
    result.addSink(new FlinkKafkaProducer[(String,Int)]("wc",new
KafkaSerializationSchema[(String, Int)] {
      override def serialize(element: (String, Int), timestamp: lang.Long):
ProducerRecord[Array[Byte], Array[Byte]] = {
        new ProducerRecord("wc",element._1.getBytes(),
(element._2+"").getBytes())
      }
    },props,FlinkKafkaProducer.Semantic.EXACTLY_ONCE))

    env.execute()
  }
}
```

MySQL Sink（幂等性）

Flink处理结果写入到MySQL中，这并不是Flink默认支持的，需要添加MySQL的驱动依赖

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.44</version>
        </dependency>
```

因为不是内嵌支持的，所以需要基于RichSinkFunction自定义sink

不要基于SinkFunction自定义sink why？看源码

消费kafka中数据，统计各个卡口的流量，并且存入到MySQL中

注意点：需要去重，操作MySQL需要幂等性

```
import java.sql.{Connection, DriverManager, PreparedStatement}
import java.util.Properties
```

```scala
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.functions.sink.{RichSinkFunction,
SinkFunction}
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.connectors.kafka.{FlinkKafkaConsumer,
KafkaDeserializationSchema}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringSerializer

object MySQLSink {

  case class CarInfo(monitorId: String, carId: String, eventTime: String, Speed:
Long)

  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers",
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    //第一个参数 ： 消费的topic名
    val stream = env.addSource(new FlinkKafkaConsumer[(String, String)]("flink-
kafka", new KafkaDeserializationSchema[(String, String)] {
      //什么时候停止，停止条件是什么
      override def isEndOfStream(t: (String, String)): Boolean = false

      //要进行序列化的字节流
      override def deserialize(consumerRecord: ConsumerRecord[Array[Byte],
Array[Byte]]): (String, String) = {
        val key = new String(consumerRecord.key(), "UTF-8")
        val value = new String(consumerRecord.value(), "UTF-8")
        (key, value)
      }

      //指定一下返回的数据类型  Flink提供的类型
      override def getProducedType: TypeInformation[(String, String)] = {
        createTuple2TypeInformation(createTypeInformation[String],
createTypeInformation[String])
      }
    }, props))

    stream.map(data => {
      val value = data._2
      val splits = value.split("\t")
      val monitorId = splits(0)
      (monitorId, 1)
    }).keyBy(_._1)
```

```
      .reduce(new ReduceFunction[(String, Int)] {
        //t1:上次聚合完的结果  t2:当前的数据
        override def reduce(t1: (String, Int), t2: (String, Int)): (String, Int)
= {
          (t1._1, t1._2 + t2._2)
        }
      }).addSink(new MySQLCustomSink)

    env.execute()
  }

  //幂等性写入外部数据库MySQL
  class MySQLCustomSink extends RichSinkFunction[(String, Int)] {
    var conn: Connection = _
    var insertPst: PreparedStatement = _
    var updatePst: PreparedStatement = _

    //每来一个元素都会调用一次
    override def invoke(value: (String, Int), context: SinkFunction.Context[_]):
Unit = {
      println(value)
      updatePst.setInt(1, value._2)
      updatePst.setString(2, value._1)
      updatePst.execute()
      println(updatePst.getUpdateCount)
      if(updatePst.getUpdateCount == 0){
        println("insert")
        insertPst.setString(1, value._1)
        insertPst.setInt(2, value._2)
        insertPst.execute()
      }
    }

    //thread初始化的时候执行一次
    override def open(parameters: Configuration): Unit = {
      conn = DriverManager.getConnection("jdbc:mysql://node01:3306/test",
"root", "123123")
      insertPst = conn.prepareStatement("INSERT INTO car_flow(monitorId,count)
VALUES(?,?)")
      updatePst = conn.prepareStatement("UPDATE car_flow SET count = ? WHERE
monitorId = ?")
    }

    //thread关闭的时候 执行一次
    override def close(): Unit = {
      insertPst.close()
      updatePst.close()
      conn.close()
    }
  }

}
```

Socket Sink

Flink处理结果发送到套接字（Socket）

基于RichSinkFunction自定义sink

```
import java.io.PrintStream
import java.net.{InetAddress, Socket}
import java.util.Properties

import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.functions.sink.{RichSinkFunction,
SinkFunction}
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment,
createTuple2TypeInformation, createTypeInformation}
import org.apache.flink.streaming.connectors.kafka.{FlinkKafkaConsumer,
KafkaDeserializationSchema}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringSerializer

//sink 到 套接字 socket
object SocketSink {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers",
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    //第一个参数 ： 消费的topic名
    val stream = env.addSource(new FlinkKafkaConsumer[(String, String)]("flink-
kafka", new KafkaDeserializationSchema[(String, String)] {
      //什么时候停止，停止条件是什么
      override def isEndOfStream(t: (String, String)): Boolean = false

      //要进行序列化的字节流
      override def deserialize(consumerRecord: ConsumerRecord[Array[Byte],
Array[Byte]]): (String, String) = {
        val key = new String(consumerRecord.key(), "UTF-8")
        val value = new String(consumerRecord.value(), "UTF-8")
        (key, value)
      }

      //指定一下返回的数据类型  Flink提供的类型
      override def getProducedType: TypeInformation[(String, String)] = {
        createTuple2TypeInformation(createTypeInformation[String],
createTypeInformation[String])
      }
    }, props))

    stream.map(data => {
      val value = data._2
      val splits = value.split("\t")
      val monitorId = splits(0)
      (monitorId, 1)
```

```
    }).keyBy(_._1)
      .reduce(new ReduceFunction[(String, Int)] {
        //t1:上次聚合完的结果  t2:当前的数据
        override def reduce(t1: (String, Int), t2: (String, Int)): (String, Int)
= {
          (t1._1, t1._2 + t2._2)
        }
      }).addSink(new SocketCustomSink("node01",8888))

    env.execute()
  }

  class SocketCustomSink(host:String,port:Int) extends
RichSinkFunction[(String,Int)]{
    var socket: Socket  = _
    var writer:PrintStream = _

    override def open(parameters: Configuration): Unit = {
      socket = new Socket(InetAddress.getByName(host), port)
      writer = new PrintStream(socket.getOutputStream)
    }

    override def invoke(value: (String, Int), context: SinkFunction.Context[_]):
Unit = {
      writer.println(value._1 + "\t" +value._2)
      writer.flush()
    }

    override def close(): Unit = {
      writer.close()
      socket.close()
    }
  }
}
```

File Sink

Flink处理的结果保存到文件，这种使用方式不是很常见

支持分桶写入，每一个桶就是一个目录，默认每隔一个小时会产生一个分桶，每个桶下面会存储每一个Thread的处理结果，可以设置一些文件滚动的策略（文件打开、文件大小等），防止出现大量的小文件，代码中详解

Flink默认支持，导入连接文件的连接器依赖

```
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-filesystem_2.11</artifactId>
            <version>1.9.2</version>
        </dependency>
```

```
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.serialization.SimpleStringEncoder
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.core.fs.Path
import
org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink
```

```scala
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.Default
RollingPolicy
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment,
createTuple2TypeInformation, createTypeInformation}
import org.apache.flink.streaming.connectors.kafka.{FlinkKafkaConsumer,
KafkaDeserializationSchema}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringSerializer

object FileSink {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers",
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    //第一个参数 ： 消费的topic名
    val stream = env.addSource(new FlinkKafkaConsumer[(String, String)]("flink-
kafka", new KafkaDeserializationSchema[(String, String)] {
      //什么时候停止，停止条件是什么
      override def isEndOfStream(t: (String, String)): Boolean = false

      //要进行序列化的字节流
      override def deserialize(consumerRecord: ConsumerRecord[Array[Byte],
Array[Byte]]): (String, String) = {
        val key = new String(consumerRecord.key(), "UTF-8")
        val value = new String(consumerRecord.value(), "UTF-8")
        (key, value)
      }

      //指定一下返回的数据类型  Flink提供的类型
      override def getProducedType: TypeInformation[(String, String)] = {
        createTuple2TypeInformation(createTypeInformation[String],
createTypeInformation[String])
      }
    }, props))

    val restStream = stream.map(data => {
      val value = data._2
      val splits = value.split("\t")
      val monitorId = splits(0)
      (monitorId, 1)
    }).keyBy(_._1)
      .reduce(new ReduceFunction[(String, Int)] {
        //t1:上次聚合完的结果  t2:当前的数据
        override def reduce(t1: (String, Int), t2: (String, Int)): (String, Int)
= {
          (t1._1, t1._2 + t2._2)
        }
      }).map(x=>x._1 + "\t" + x._2)
```

```
        //设置文件滚动策略
    val rolling:DefaultRollingPolicy[String,String] =
DefaultRollingPolicy.create()
        //当文件超过2s没有写入新数据，则滚动产生一个小文件
        .withInactivityInterval(2000)
        //文件打开时间超过2s 则滚动产生一个小文件  每隔2s产生一个小文件
        .withRolloverInterval(2000)
        //当文件大小超过256 则滚动产生一个小文件
        .withMaxPartSize(256*1024*1024)
        .build()

    /**
      * 默认:
      * 每一个小时对应一个桶（文件夹），每一个thread处理的结果对应桶下面的一个小文件
      * 当小文件大小超过128M或者小文件打开时间超过60s,滚动产生第二个小文件
      */
     val sink: StreamingFileSink[String] = StreamingFileSink.forRowFormat(
      new Path("d:/data/rests"),
      new SimpleStringEncoder[String]("UTF-8"))
         .withBucketCheckInterval(1000)
         .withRollingPolicy(rolling)
         .build()

//    val sink = StreamingFileSink.forBulkFormat(
//      new Path("./data/rest"),
//      ParquetAvroWriters.forSpecificRecord(classOf[String])
//    ).build()

    restStream.addSink(sink)
    env.execute()
  }
}
```

HBase Sink

计算结果写入sink 两种实现方式:

1. map算子写入 频繁创建hbase连接
2. process写入 适合批量写入hbase

导入HBase依赖包

```
          <dependency>
              <groupId>org.apache.hbase</groupId>
              <artifactId>hbase-client</artifactId>
              <version>${hbase.version}</version>
          </dependency>
          <dependency>
              <groupId>org.apache.hbase</groupId>
              <artifactId>hbase-common</artifactId>
              <version>${hbase.version}</version>
          </dependency>
          <dependency>
              <groupId>org.apache.hbase</groupId>
              <artifactId>hbase-server</artifactId>
              <version>${hbase.version}</version>
          </dependency>
```

读取kafka数据，统计卡口流量保存至HBase数据库中

1. HBase中创建对应的表

```
create 'car_flow',{NAME => 'count', VERSIONS => 1}
```

2. 实现代码

```
import java.util.{Date, Properties}

import com.msb.stream.util.{DateUtils, HBaseUtil}
import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.functions.ProcessFunction
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.flink.util.Collector
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.{HTable, Put}
import org.apache.hadoop.hbase.util.Bytes
import org.apache.kafka.common.serialization.StringSerializer


object HBaseSinkTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元数据）
    props.setProperty("bootstrap.servers", "node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new SimpleStringSchema(), props))


    stream.map(row => {
      val arr = row.split("\t")
      (arr(0), 1)
    }).keyBy(_._1)
      .reduce((v1: (String, Int), v2: (String, Int)) => {
        (v1._1, v1._2 + v2._2)
      }).process(new ProcessFunction[(String, Int), (String, Int)] {

      var htab: HTable = _

      override def open(parameters: Configuration): Unit = {
        val conf = HBaseConfiguration.create()
        conf.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:2181")
```

```
        val hbaseName = "car_flow"
        htab = new HTable(conf, hbaseName)
      }

      override def close(): Unit = {
        htab.close()
      }

      override def processElement(value: (String, Int), ctx:
ProcessFunction[(String, Int), (String, Int)]#Context, out: Collector[(String,
Int)]): Unit = {
        // rowkey:monitorid   时间戳（分钟） value：车流量
        val min = DateUtils.getMin(new Date())
        val put = new Put(Bytes.toBytes(value._1))
        put.addColumn(Bytes.toBytes("count"), Bytes.toBytes(min),
Bytes.toBytes(value._2))
        htab.put(put)
      }
    })
    env.execute()
  }
}
```

Flink Window操作

Flink任务Batch是Streaming的一个特例，因此Flink底层引擎是一个流式引擎，在上面实现了流处理和批处理。而Window就是从Streaming到Batch的桥梁

Window窗口就在一个无界流中设置起始位置和终止位置，让无界流变成有界流，并且在有界流中进行数据处理

Window操作常见的业务场景：统计过去一段时间、最近一些元素的数据指标

Window窗口分类

Window窗口在无界流中设置起始位置和终止位置的方式可以有两种：

- 根据时间设置
- 根据窗口数据量（count）设置

根据窗口的类型划分：

- 滚动窗口
- 滑动窗口

根据数据流类型划分：

- Keyed Window：基于分组后的数据流之上做窗口操作

- Global Window：基于未分组的数据流之上做窗口操作

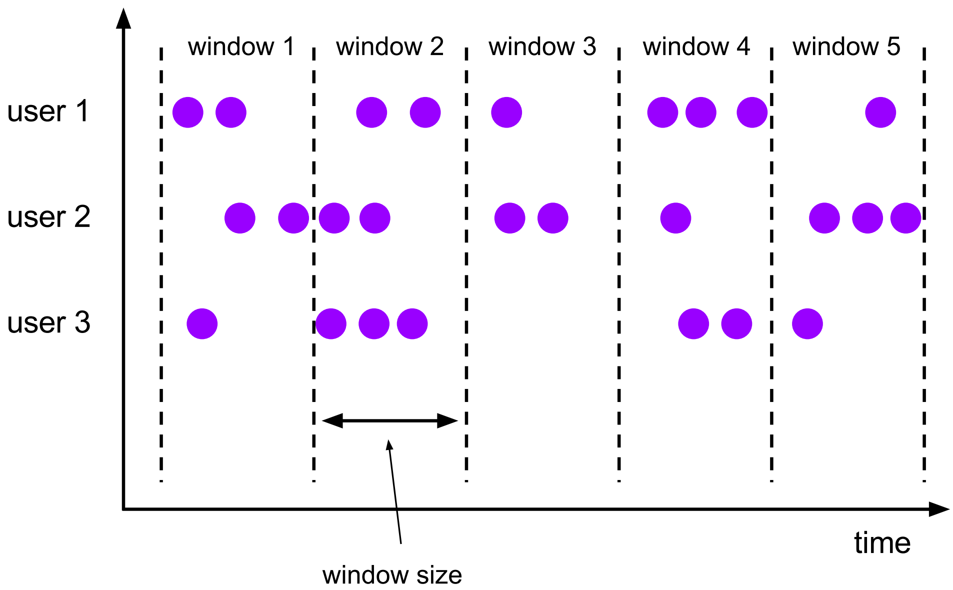
根据不同的组合方式，可以组合出来8种窗口类型：

1. 基于分组后的数据流上的时间滚动窗口
2. 基于分组后的数据流上的时间滑动窗口
3. 基于分组后的数据流上的count滚动窗口
4. 基于分组后的数据流上的count滑动窗口
5. 基于未分组的数据流上的时间滚动窗口
6. 基于未分组的数据流上的时间滑动窗口

7. 基于未分组的数据流上的count滚动窗口
8. 基于未分组的数据流上的count滑动窗口

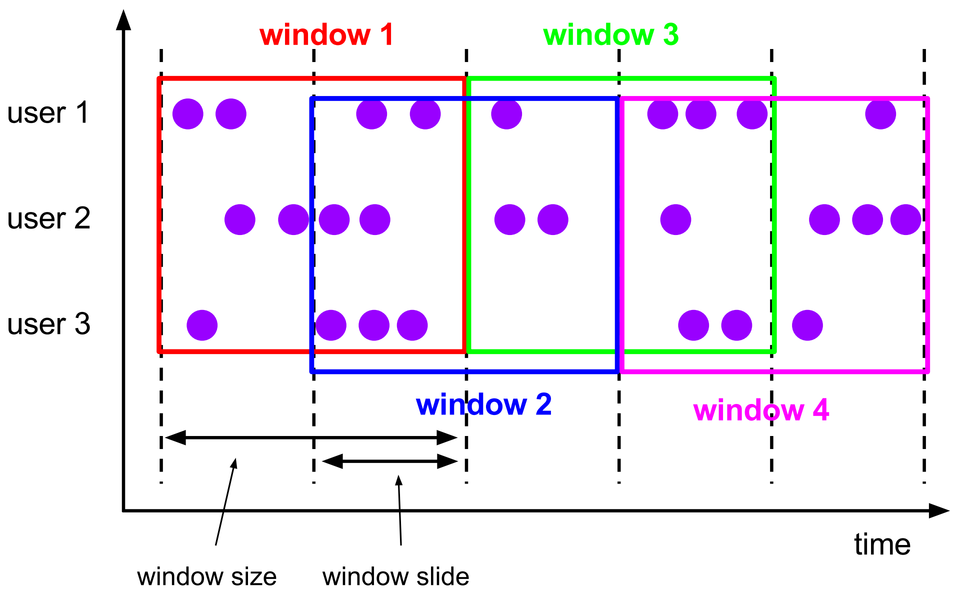
当然我们也可以根据实际业务场景自定义Window，这就是Flink最大的优势：Window种类多，灵活

- Time Window（基于时间的窗口）
  - Tumbling Window：滚动窗口，窗口之间没有数据重叠



  - Sliding Window：滑动窗口，窗口内的数据有重叠

    在定义滑动窗口的时候，不只是要定义窗口大小，还要定义窗口的滑动间隔时间（每隔多久滑动一次），如果滑动间隔时间=窗口大小=滚动窗口



窗口函数

窗口函数定义了针对窗口内元素的计算逻辑，窗口函数大概分为两类：

1. 增量聚合函数，聚合原理：窗口内保存一个中间聚合结果，随着新元素的加入，不断对该值进行更新

   这类函数通常非常节省空间 ReduceFunction、AggregateFunction属于增量聚合函数

2. 全量聚合函数，聚合原理：收集窗口内的所有元素，并且在执行的时候对他们进行遍历，这种聚合函数通常需要占用更多的空间（收集一段时间的数据并且保存），但是它可以支持更复杂的逻辑 ProcessWindowFunction、WindowFunction属于全量窗口函数

   注意：这两类函数可以组合搭配使用

案例1：使用增量聚合函数统计最近20s内，各个卡口的车流量

```
import java.util.Properties

import org.apache.flink.api.common.functions.AggregateFunction
import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment,
createTypeInformation}
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.flink.util.Collector
import org.apache.kafka.common.serialization.StringSerializer

/**
  * 使用增量聚合函数统计最近20s内，各个卡口的车流量
  */
object Demo01StatisCarFlow {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers",
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new
SimpleStringSchema(), props))

    //monitorId + "\t").append(carId + "\t").append(timestamp +
"\t").append(speed)
    stream.map(data => {
      val arr = data.split("\t")
      val monitorID = arr(0)
      (monitorID, 1)
    }).keyBy(_._1)
      .timeWindow(Time.seconds(10))
      //      .reduce(new ReduceFunction[(String, Int)] {
      //        override def reduce(value1: (String, Int), value2: (String,
Int)): (String, Int) = {
      //          (value1._1, value1._2 + value2._2)
      //        }
      //      }).print()
      .aggregate(new AggregateFunction[(String, Int), Int, Int] {
      override def createAccumulator(): Int = 0
```

```
      override def add(value: (String, Int), acc: Int): Int = acc + value._2

      override def getResult(acc: Int): Int = acc

      override def merge(a: Int, b: Int): Int = a + b
    },
//      new WindowFunction[Int, (String, Int), String, TimeWindow] {
//      override def apply(key: String, window: TimeWindow, input: 
Iterable[Int], out: Collector[(String, Int)]): Unit = {
//        for (elem <- input) {
//          out.collect((key, elem))
//        }
//      }
//    }
    new ProcessWindowFunction[Int, (String, Int), String, TimeWindow] {
      override def process(key: String, context: Context, elements: 
Iterable[Int], out: Collector[(String, Int)]): Unit = {
        for (elem <- elements) {
          out.collect((key,elem))
        }
      }
    }
    ).print()
    env.execute()
  }
}
```

ProcessWindowFunction、WindowFunction区别在于ProcessWindowFunction可以获取Flink执行的上下文，可以拿到当前的数据更多信息，比如窗口状态、当前水印、时间戳等

案例2：每隔10s统计每辆汽车的平均速度

```
import java.util.Properties

import org.apache.flink.api.common.functions.AggregateFunction
import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.kafka.common.serialization.StringSerializer

object Demo03SpeedAVG {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers", 
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)
```

```
    val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new
SimpleStringSchema(), props))

    stream.map(data => {
      val splits = data.split("\t")
      (splits(1),splits(3).toInt)
    }).keyBy(_._1)
      .timeWindow(Time.seconds(10))
      .aggregate(new AggregateFunction[(String,Int),(String,Int,Int),
(String,Double)] {
        override def createAccumulator(): (String, Int, Int) = ("",0,0)

        override def add(value: (String, Int), accumulator: (String, Int, Int)):
(String, Int, Int) = {
          (value._1,value._2+accumulator._2,accumulator._3+1)
        }

        override def getResult(accumulator: (String, Int, Int)): (String,
Double) = {
          (accumulator._1,accumulator._2.toDouble/accumulator._3)
        }

        override def merge(a: (String, Int, Int), b: (String, Int, Int)):
(String, Int, Int) = {
          (a._1,a._2+b._2,a._3+b._3)
        }
      }).print()

    env.execute()
  }
}
```

案例3：每隔10s对窗口内所有汽车的车速进行排序

```
import java.util.Properties

import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.scala.function.ProcessAllWindowFunction
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.flink.util.Collector
import org.apache.kafka.common.serialization.StringSerializer

object Demo02SortSpeed {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers",
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
```

```
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new
SimpleStringSchema(), props))

    stream.map(data => {
      val splits = data.split("\t")
      (splits(1),splits(3).toInt)
    }).timeWindowAll(Time.seconds(10))
      //注意：想要全局排序并行度需要设置为1
      .process(new ProcessAllWindowFunction[(String,Int),String,TimeWindow] {
        override def process(context: Context, elements: Iterable[(String,
Int)], out: Collector[String]): Unit = {
          val sortList = elements.toList.sortBy(_._2)
          for (elem <- sortList) {
            out.collect(elem._1+" speed:" + elem._2)
          }
        }
      }).print()
    env.execute()
  }
}
```

案例4：每隔10s统计出窗口内所有车辆的最大及最小速度

```
import java.util.Properties

import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.kafka.common.serialization.StringSerializer
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.function.ProcessAllWindowFunction
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector


object Demo04MaxMinSpeed {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //设置连接kafka的配置信息
    val props = new Properties()
    //注意   sparkstreaming + kafka（0.10之前版本） receiver模式  zookeeper url（元
数据）
    props.setProperty("bootstrap.servers",
"node01:9092,node02:9092,node03:9092")
    props.setProperty("group.id", "flink-kafka-001")
    props.setProperty("key.deserializer", classOf[StringSerializer].getName)
    props.setProperty("value.deserializer", classOf[StringSerializer].getName)

    val stream = env.addSource(new FlinkKafkaConsumer[String]("flink-kafka", new
SimpleStringSchema(), props))
    stream.map(data =>{
      val arr = data.split("\t")
      (arr(1),arr(3).toInt)
    }).timeWindowAll(Time.seconds(20))
```

```
      .process(new ProcessAllWindowFunction[(String,Int),String,TimeWindow] {
        override def process(context: Context, elements: Iterable[(String,
Int)], out: Collector[String]): Unit = {
          val sortList = elements.toList.sortBy(_._2)
          println(sortList)
          val minSpeedInfo = sortList.head
          val maxSpeedInfo = sortList.last
          val startWindowTime = context.window.getStart
          val endWindowTime = context.window.getEnd
          out.collect(
          "窗口起始时间："+startWindowTime  + "结束时间："+ endWindowTime +" 最小车辆
速度车牌号：" + minSpeedInfo._1 + " 车速："+minSpeedInfo._2 + "\t最大车辆速度车牌号："
+ maxSpeedInfo._1 + " 车速：" + maxSpeedInfo._2
          )
        }
      }).print()
    env.execute()
  }
}
```