Flink超神文档

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

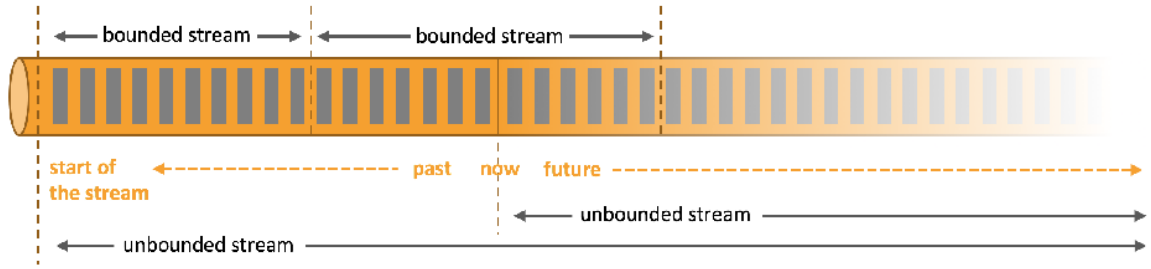
什么是Unbounded streams?

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

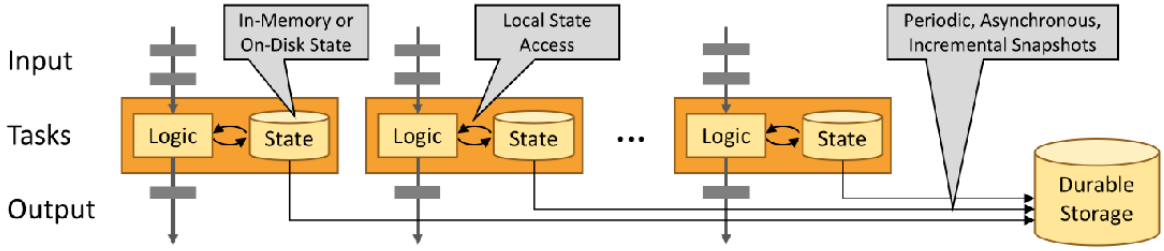


什么是stateful computations?

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

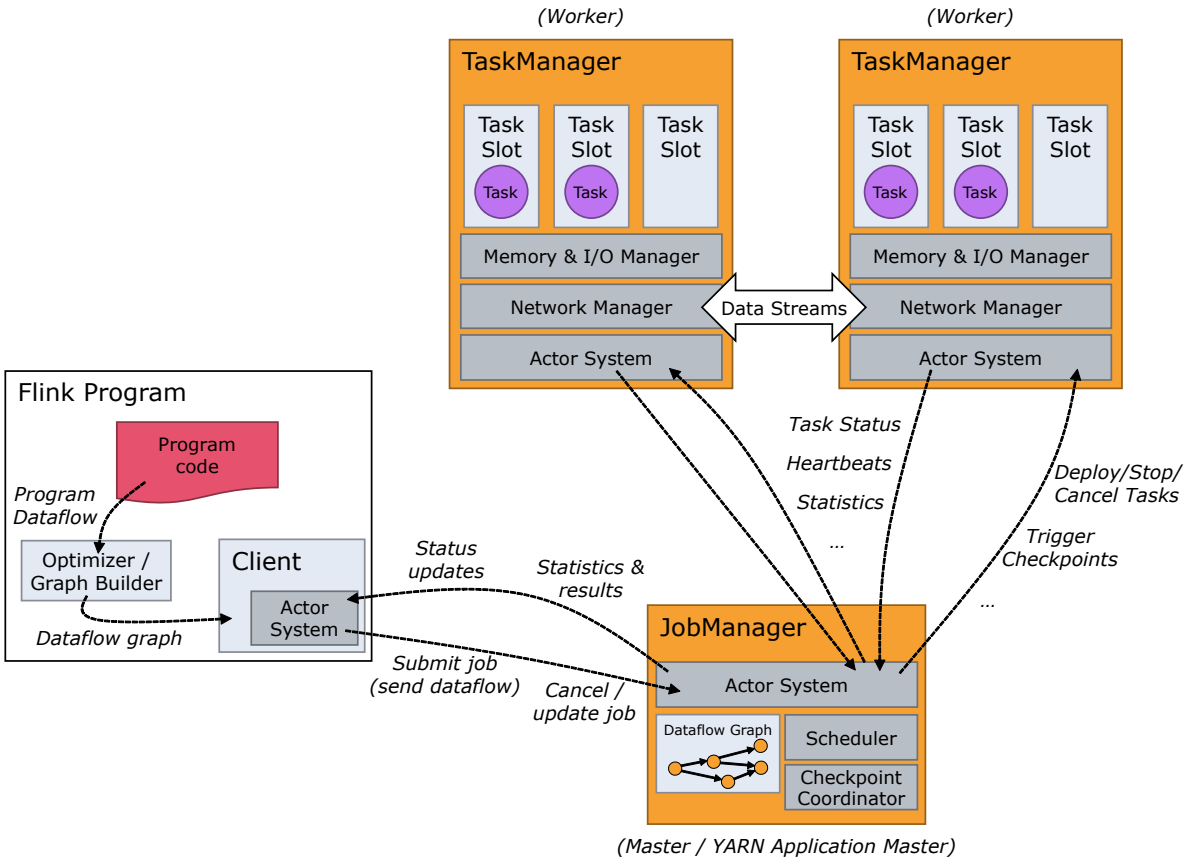
Flink的特点和优势

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
```

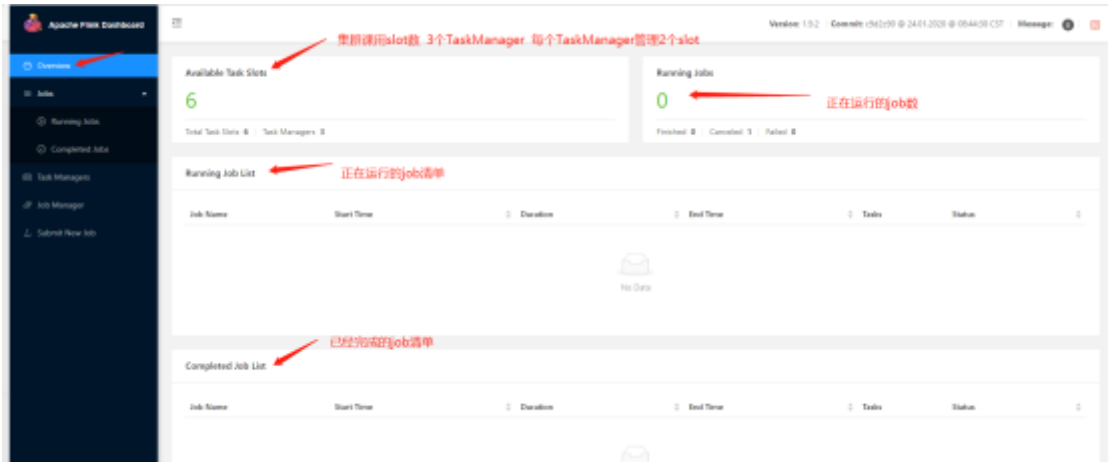
6. 启动standalone集群

   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>
```

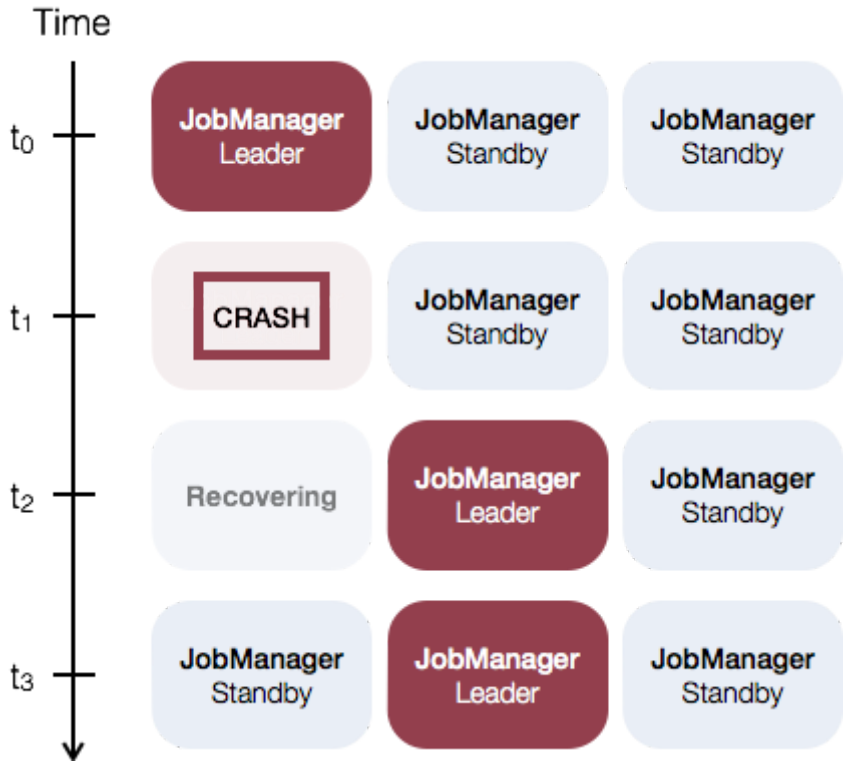
Standalone HA集群安装&测试

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

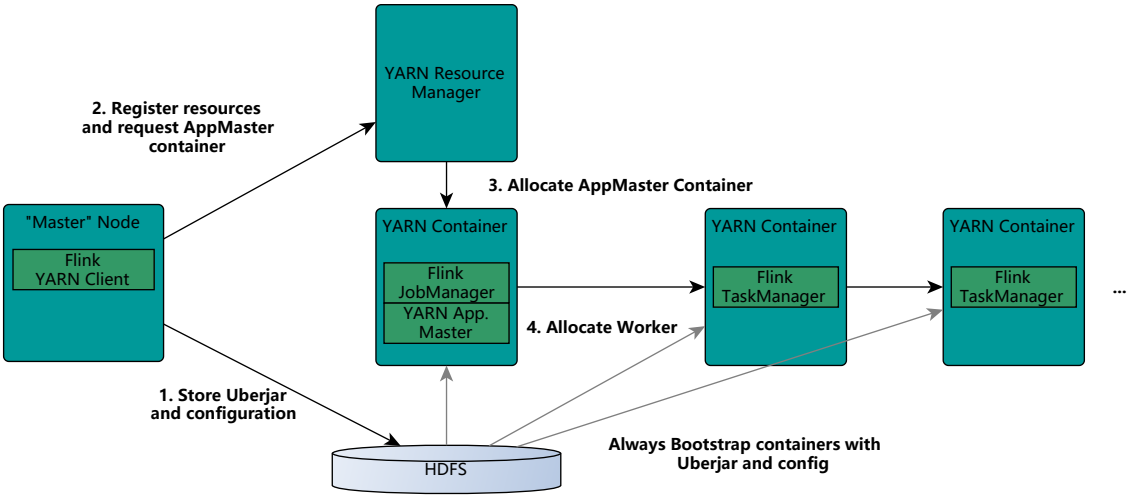
Flink on Yarn

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

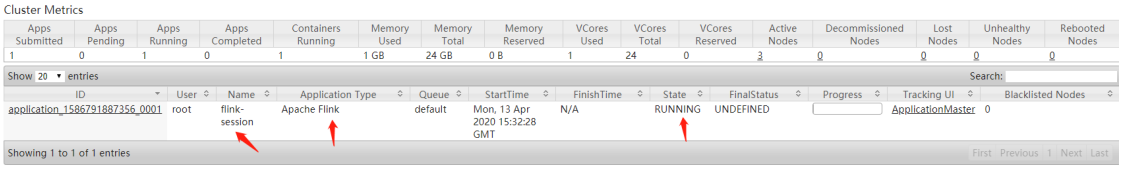
配置两种运行模式

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）
```



- 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

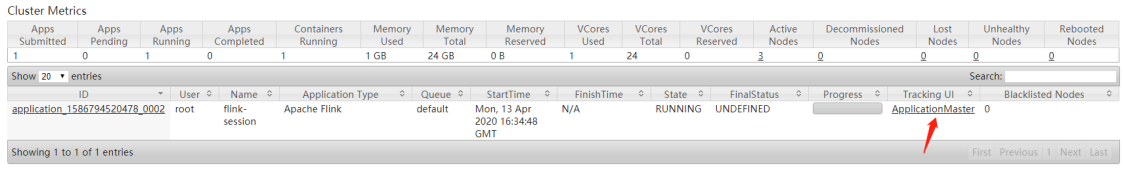
两种模式都可以测试，因为不管哪种模式都会启动yarn-session

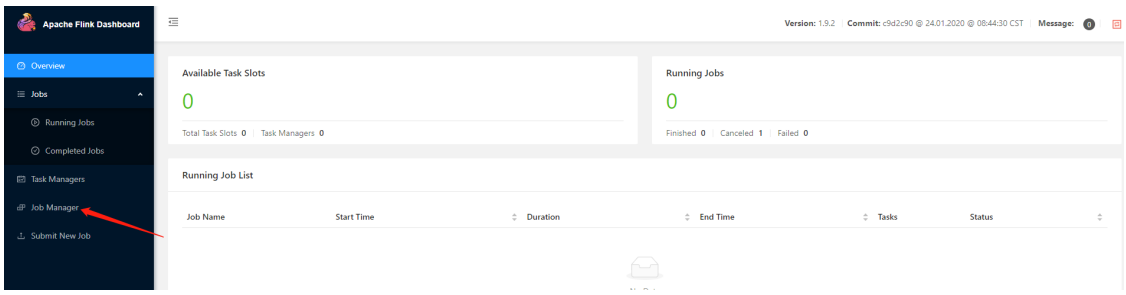
yarn-session模式测试

- 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



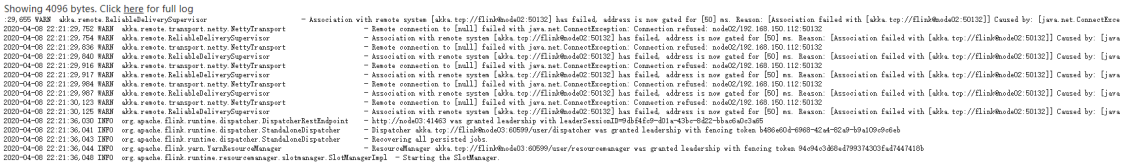


high-availability	zookeeper
high-availability.cluster-id	application_1586794520478_0002
high-availability.storageDir	hdfs://node01:9000/flink/ha/
high-availability.zookeeper.quorum	node01:2181,node02:2181,node03:2181
internal.cluster.execution-mode	NORMAL
internal.io.tmpdirs.use-local-default	true
io.tmp.dirs	/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




```
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
```

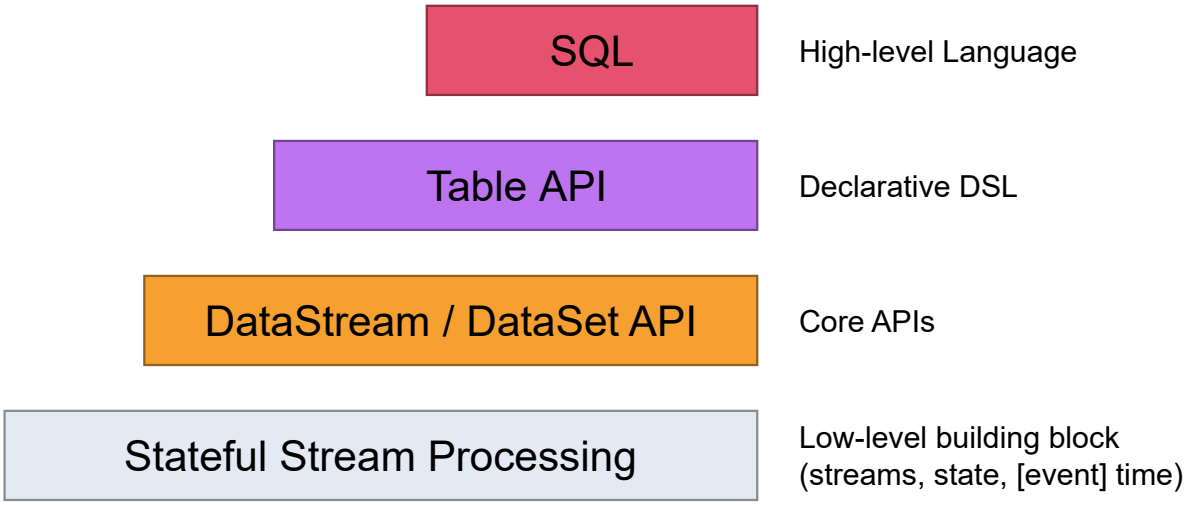
- 停掉JobManager 观察
- 测试完毕，取消job

```
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

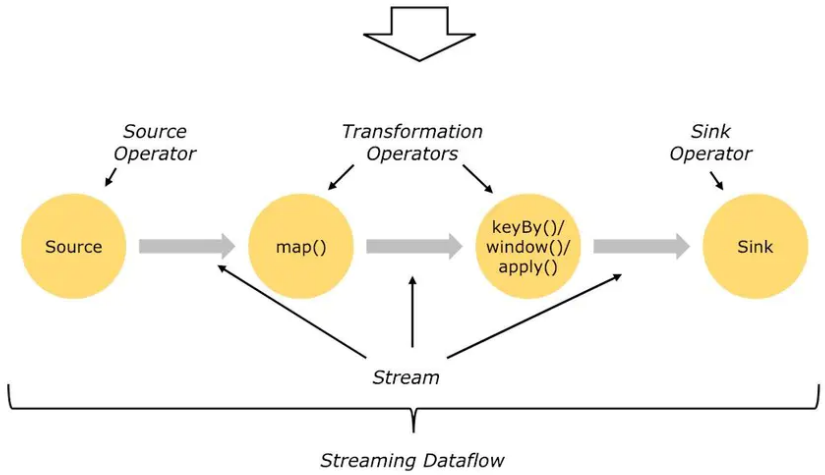
在Flink的世界观中，一切都是数据流，所以对于批计算来说，那只是流计算的一个特例而已

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.....）

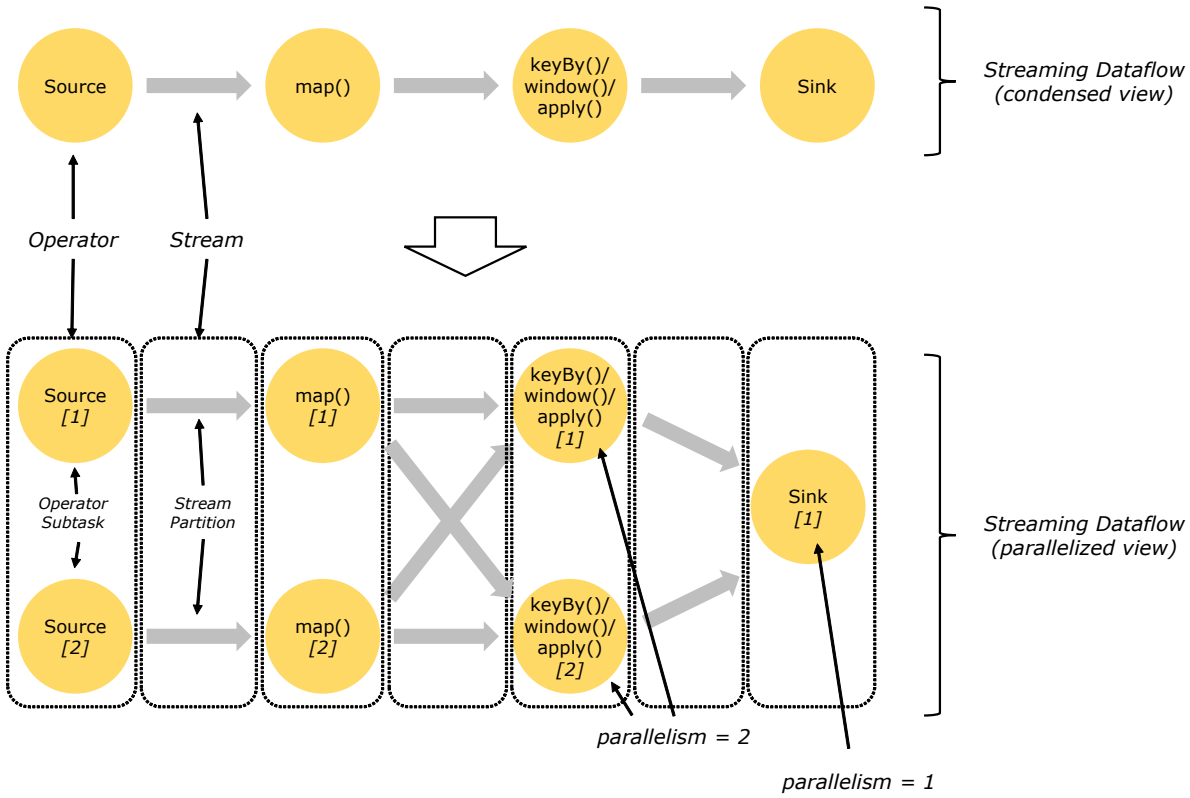
基于Flink开发的程序都能够映射成一个Dataflows

```
DataStream<String> lines = env.addSource(
                            new FlinkKafkaConsumer<>(…));      } Source

DataStream<Event> events = lines.map((line) -> parse(line));  } Transformation

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());             } Transformation

stats.addSink(new RollingSink(path));                          } Sink
```



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.ManifestResourceTransfor
mer">
                                <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
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()
    //读完就停止
    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

```
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

```
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()
```

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

DataStream 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 → 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

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

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()
```

Select

SplitStream → DataStream

从SplitStream中选择一个或者多个数据流

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