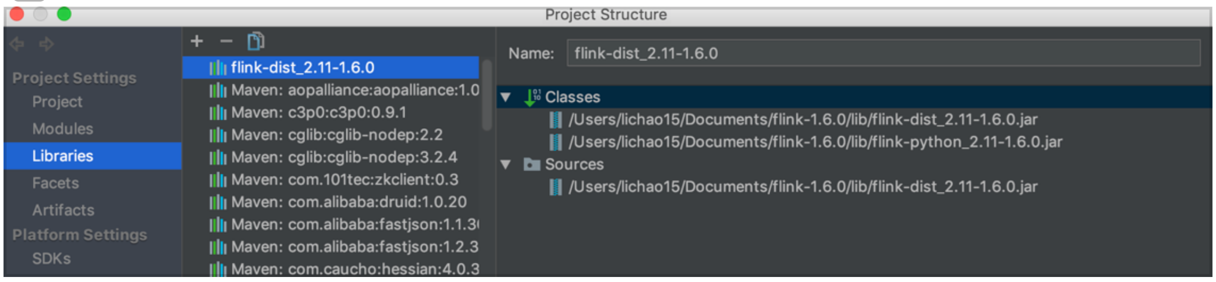
Flink学习

#学习

基础

相关概念

- [ ] flink程序能实现在分布式的结合上进行各种转换操作，集合通常来自订阅的来源（文件，kafka,local,in-memory），结果被返回到sinks里（大多数写入分布式文件系统，或者标准输出）
- [ ] flink 的operator 会尽量按照链式方式分在同一个task slot里面，以优化执行
- [ ] DataSet and DataStream
  - [ ] DataSet和DataStream在flink中都代表一种数据结构，是不可变且包含重复记录的集合。区别在于DataSet是有限的集合，而DataStream是无界的
- [ ] flink 配置interlij ideal 在本地运行调试
  - [ ] 只需要将flink依赖的包引入项目中即可启动项目



flink-dist_2.11-1.6.0
Maven: aopalliance:aopalliance:1.0
Maven: c3p0:c3p0:0.9.1
Maven: cglib:cglib-nodep:2.2
Maven: cglib:cglib-nodep:3.2.4
Maven: com.101tec:zkclient:0.3
Maven: com.alibaba:druid:1.0.20
Maven: com.alibaba:fastjson:1.1.3
Maven: com.alibaba:fastjson:1.2.3
Maven: com.caucho:hessian:4.0.3

Name: flink-dist_2.11-1.6.0

Classes
/Users/lichao15/Documents/flink-1.6.0/lib/flink-dist_2.11-1.6.0.jar
/Users/lichao15/Documents/flink-1.6.0/lib/flink-python_2.11-1.6.0.jar
Sources
/Users/lichao15/Documents/flink-1.6.0/lib/flink-dist_2.11-1.6.0.jar

- [ ] 讲解Flink怎么序列化objects，怎么分配内存Apache Flink: Juggling with Bits and Bytes

DataStream

- [ ] Apache Flink 1.7 Documentation: Flink DataStream API Programming Guide
- [ ] datasource（数据源）：
  - [ ] File-based: readTextFile, readFile…
  - [ ] Socket-based: socketTextStream
  - [ ] Collection-based: fromCollection, fromElements
  - [ ] custom: addSource, FlinkKafkaConsumer08 or other connectors

DataSet

- [ ] Apache Flink 1.7 Documentation: Flink DataSet API Programming Guide
- [ ] 不得不会的Flink Dataset的DeltaI 迭代操作

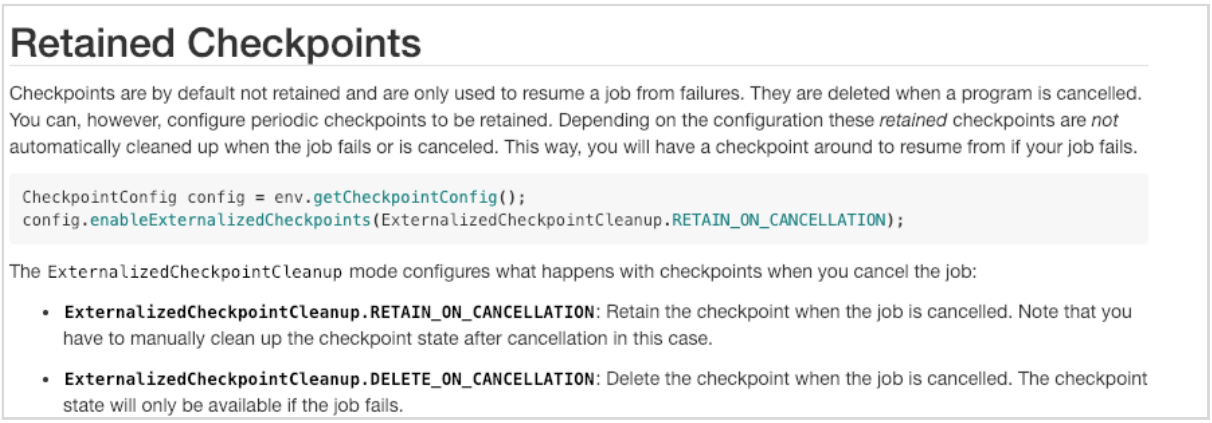
savepoint

- [ ] Apache Flink 1.7 Documentation: Savepoints
- [ ] Savepoints are created, owned, and deleted by the user.

- [ ] 目前savepoint和checkpoint实现和format方式都相同（除了checkpoint选择了rocksdb作为state backend，这样format会有些微不同）
- [ ] Operations：
  - [ ] Triggering Savepoints： FsStateBackend or RocksDBStateBackend:
  - [ ] Trigger a Savepoint
  - [ ] Cancel Job with Savepoint
    - [ ] `bin/flink cancel -s [:targetDirectory] :jobId`
  - [ ] Resuming from Savepoints
    - [ ] `$ bin/flink run -s :savepointPath [:runArgs]`
  - [ ] Disposing Savepoints
    - [ ] `$ bin/flink savepoint -d :savepointPath`

checkpoint

- [ ] Apache Flink 1.7 Documentation: Checkpoints
- [ ] 生命周期是由Flink管理，checkpoint的管理，创建以及释放统一通过Flink，而不需要用户干预
- [ ] Checkpoints are usually dropped（随应用退出被清除） after the job was terminated by the user (except if explicitly configured as retained Checkpoints)



Retained Checkpoints

Checkpoints are by default not retained and are only used to resume a job from failures. They are deleted when a program is cancelled. You can, however, configure periodic checkpoints to be retained. Depending on the configuration these *retained* checkpoints are *not* automatically cleaned up when the job fails or is canceled. This way, you will have a checkpoint around to resume from if your job fails.

```
CheckpointConfig config = env.getCheckpointConfig();
config.enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

The `ExternalizedCheckpointCleanup` mode configures what happens with checkpoints when you cancel the job:

- **`ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION`**: Retain the checkpoint when the job is cancelled. Note that you have to manually clean up the checkpoint state after cancellation in this case.
- **`ExternalizedCheckpointCleanup.DELETE_ON_CANCELLATION`**: Delete the checkpoint when the job is cancelled. The checkpoint state will only be available if the job fails.

  - [ ] retained checkpoints 不会随着应用清除，所以需要手动清理，与savepoint有一点区别，不支持rescaling
  - [ ] resuming from checkpoints

```
bin/flink run -s :checkpointMetaDataPath [:runArgs]
```

- [ ] checkpoint 优化 Apache Flink 1.7 Documentation: Tuning Checkpoints and Large State
  - [ ] state 双写：一份在distributed storage(HDFS)；一份在local
  - [ ] task-local recovery：默认是关闭的状态,可以通过 `state.backend.local-recovery` 打开

Barriers

- [ ] Apache Flink 1.8-SNAPSHOT Documentation: Data Streaming Fault Tolerance

Window，waterMark，Trigger

- [ ] Window，waterMark，Trigger介绍- 简书
- [ ] window
  - [ ] 非常重要还未研究 Windows
  - [ ] 滚动窗口：分配器将每个元素分配到一个指定窗口大小的窗口中，并且不会重叠；TumblingEventTimeWindows.of(Time.seconds(5))
  - [ ] 滑动窗口：滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率；因此可能出现窗口重叠，如果滑动参数小于滚动参数的话；SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5))
  - [ ] 会话窗口：通过session活动来对元素进行分组，跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况。当他在一个固定的时间周期内不再收到元素，即非活动间隔产生，那么窗口就会关闭；
    - [ ] 一个session窗口通过一个session间隔来配置，这个session间隔定义了非活跃周期的长度。当这个非活跃周期产生，那么当前的session将关闭并且后续的元素将被分配到新的session窗口中去。如：EventTimeSessionWindows.withGap(Time.minutes(10)
- [ ] 触发器(Triggers)
  - [ ] 触发器定义了一个窗口何时被窗口函数处理
  - [ ] EventTimeTrigger
  - [ ] ProcessingTimeTrigger
  - [ ] CountTrigger
  - [ ] PurgingTrigger
- [ ] 驱逐器(Evictors)

任务提交与停止姿势

- [ ] 任务提交
  - [ ] 启动命令详解 :Apache Flink 1.7 Documentation: YARN Setup
  - [ ] 参数

```
Usage:
   Required
     -n,--container <arg>   Number of YARN container to allocate (=Number of
Task Managers)
   Optional
     -D <arg>                        Dynamic properties
     -d,--detached                   Start detached
     -jm,--jobManagerMemory <arg>    Memory for JobManager Container with
optional unit (default: MB)
     -nm,--name                      Set a custom name for the application on
YARN
     -q,--query                      Display available YARN resources (memory,
cores)
     -qu,--queue <arg>               Specify YARN queue.
     -s,--slots <arg>                Number of slots per TaskManager
     -tm,--taskManagerMemory <arg>   Memory per TaskManager Container with
optional unit (default: MB)
     -z,--zookeeperNamespace <arg>   Namespace to create the Zookeeper sub-
paths for HA mode
```

- [ ] 提交到yarn-cluster上需要以 y 或者yarn作为前缀；如: `ynm=nm`

```
flink run -c com.jacobs.jobs.realtime.wordcount.WindowWordCount target/real-
time-jobs-1.0.0-SNAPSHOT.jar

flink run -m yarn-cluster -ynm SinkToKafkaStream -yn 4 -yjm 1024m -ytm 4096m -
ys 4 -yqu feed.prod -c com.weibo.api.feed.dm.stream.TestFlinkStream /data1/dm-
flink/feed-dm-flink-1.0.4-SNAPSHOT.jar

flink run -m yarn-cluster -ynm SinkToKafkaStream -yn 2 -yjm 1024m -ytm 4096m -
ys 2 -yqu feed.prod -c com.weibo.api.feed.dm.stream.SinkToKafkaStream /data1/
dm-flink/feed-dm-flink-1.0.4-SNAPSHOT.jar
```

- [ ] 停止任务
  - [ ] 关闭或重启flink程序不能直接kill掉，这样会导致flink来不及制作checkpoint，而应该调用flink提供的cancel语意

```
//重启正确姿势, with savepoint
1. 调用cancel, cancel之前先触发savepoint
bin/flink cancel -s [:targetDirectory] :jobId -yid: yarnAppId
例子: flink cancel -s hdfs://vcp-yz-nameservice1/user/hcp/hcpsys/feed/flink-
checkpoints/test-user-logs 97b4e67859af4bfb1b597355f1c846f3 -yid
application_1542801635735_2121
2. 从savepoint中恢复flink程序
```

```
bin/flink run -s :savepointPath [:runArgs]
例子: flink run -s hdfs://vcp-yz-nameservice1/user/hcp/hcpsys/feed/flink-
checkpoints/test-user-logs/savepoint-97b4e6-22dd5890dd0c -m yarn-cluster -ynm
TestSinkUserLogStream -yn 4 -yjm 1024m -ytm 4096m -ys 4 -yqu feed.prod -c
com.weibo.api.feed.dm.stream.TestFlinkStream /data1/dm-flink/feed-dm-
flink-1.0.4-SNAPSHOT.jar

3. 查看运行中的任务
flink list -m yarn-cluster -yid application_1548467718478_0002 -r
```

运行模式

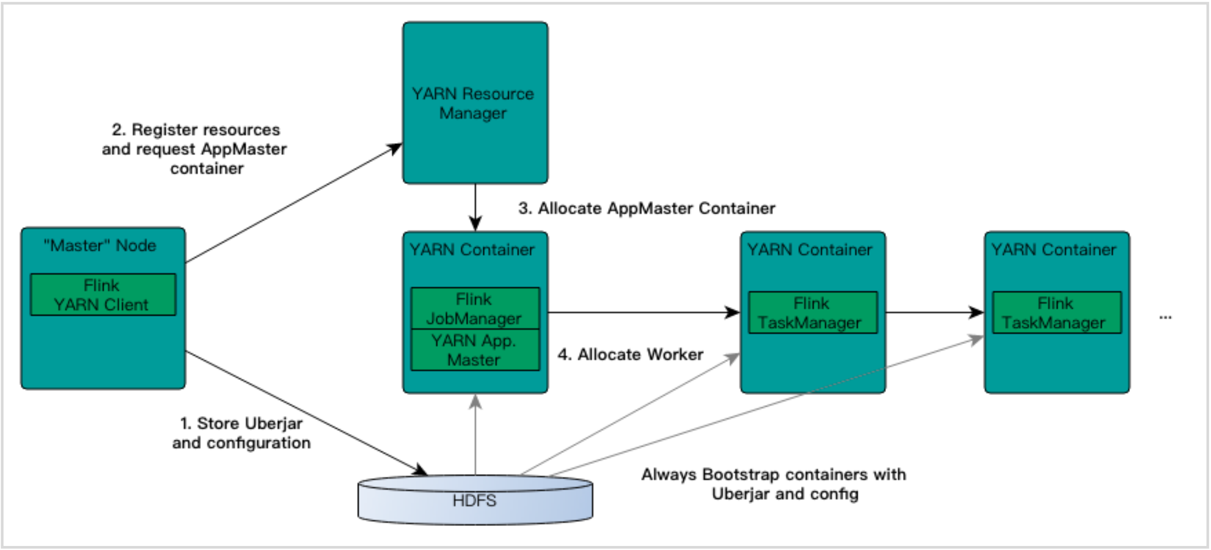
Standalone

- [ ] standalone 启动cluster

```
/usr/local/flink-1.6.0/bin;./start-cluster.sh
```

On Yarn Cluster

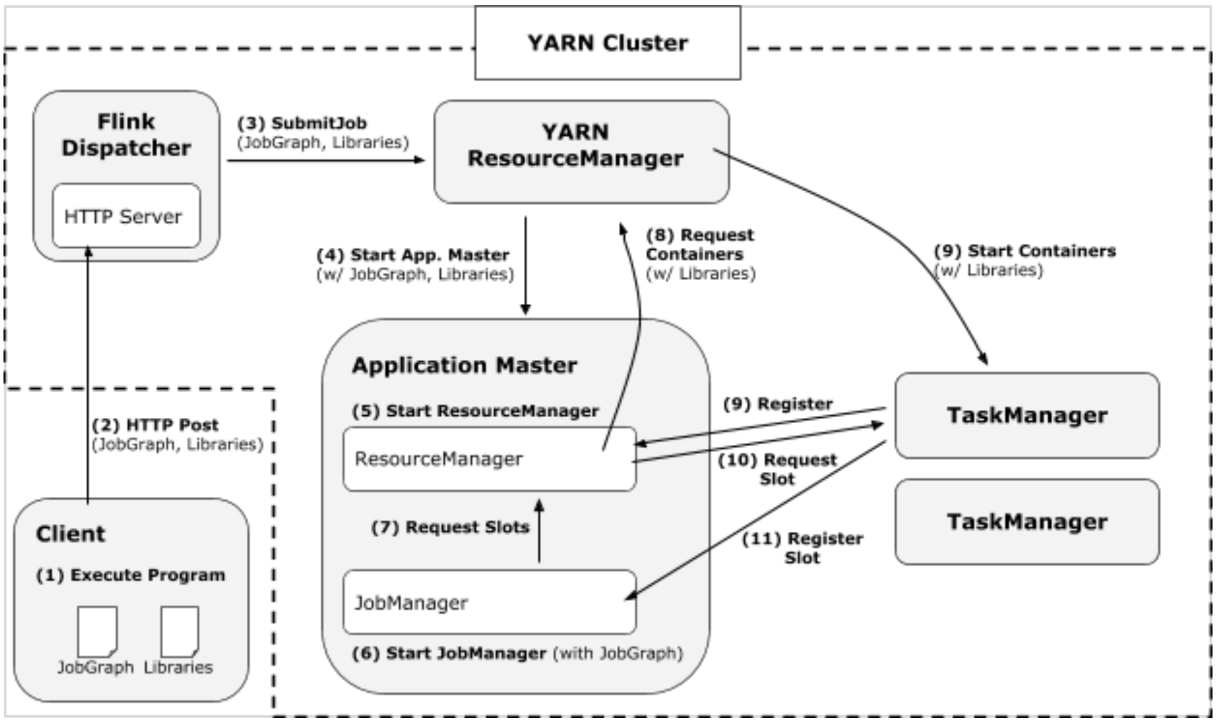
- [ ] Apache Flink 1.7 Documentation: YARN
- [ ] 参考文章Flink1.6 - flink on yarn分布式部署架构 - 深山含笑
- [ ] 架构图



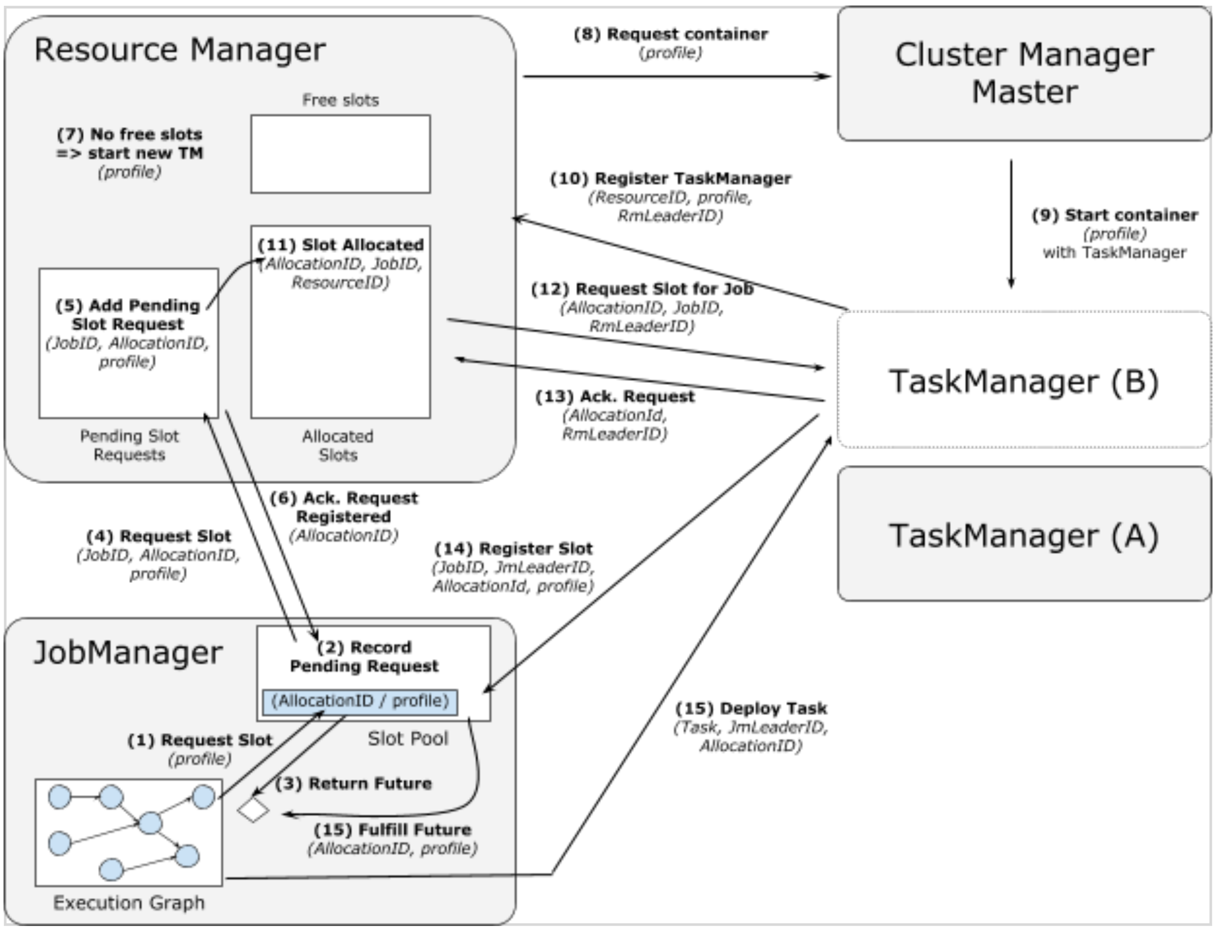
  - [ ] JobManager 和 ApplicationMaster 运行在同一个JVM里

- [ ] on yarn 两种模式
  - [ ] session模式：允许运行多个作业在同一个Flink集群中，代价是作业之间没有资源隔离（同一个TM中可能跑多个不同作业的task）

- [ ] per-job模式（生产环境）：per-job模式是指在yarn上为每一个Flink作业都分配一个单独的Flink集群，这样就解决了不同作业之间的资源隔离问题
- [ ] 摘录参考文章 相比旧的Flink-on-YARN架构（Flink 1.5之前），新的yarn架构带来了以下的优势：
  - [ ] client可以直接在yarn上面启动一个作业，不在像以前需要先启动一个固定大小的Flink集群然后把作业提交到这个Flink集群上
  - [ ] 按需申请容器（指被同一个作业的不同算子所使用的容器可以有不同的CPU/Memory配置），没有被使用的容器将会被释放


YARN Cluster
Flink Dispatcher
HTTP Server
(3) SubmitJob (JobGraph, Libraries)
YARN ResourceManager
(4) Start App. Master (w/ JobGraph, Libraries)
(8) Request Containers (w/ Libraries)
(9) Start Containers (w/ Libraries)
Application Master
(5) Start ResourceManager
ResourceManager
(7) Request Slots
JobManager
(6) Start JobManager (with JobGraph)
(9) Register
(10) Request Slot
(11) Register Slot
TaskManager
TaskManager
(2) HTTP Post (JobGraph, Libraries)
Client
(1) Execute Program
JobGraph
Libraries

- [ ] slot资源申请/分配流程分析
- [ ] 请求新TaskExecutor的slot分配


Resource Manager
Free slots
(7) No free slots => start new TM (profile)
(11) Slot Allocated (AllocationID, JobID, ResourceID)
(5) Add Pending Slot Request (JobID, AllocationID, profile)
Pending Slot Requests
Allocated Slots
(8) Request container (profile)
Cluster Manager Master
(10) Register TaskManager (ResourceID, profile, RmLeaderID)
(9) Start container (profile) with TaskManager
(12) Request Slot for Job (AllocationID, JobID, RmLeaderID)
TaskManager (B)
(13) Ack. Request (AllocationId, RmLeaderID)
TaskManager (A)
(6) Ack. Request Registered (AllocationID)
(4) Request Slot (JobID, AllocationID, profile)
(14) Register Slot (JobID, JmLeaderID, AllocationId, profile)
JobManager
(2) Record Pending Request
(AllocationID / profile)
Slot Pool
(1) Request Slot (profile)
(3) Return Future
(15) Fulfill Future (AllocationID, profile)
(15) Deploy Task (Task, JmLeaderID, AllocationID)
Execution Graph

- [ ] ResourceManager挂掉： 不会挂掉task,不断尝试重新注册ResourceManager详细见参考文章
- [ ] TaskExecutor挂掉
- [ ] JobMaster挂掉

资源分配相关？

- [ ] 在operator中对并行度的设置将决定任务分配到几个task slot里面去
- [ ] 申请资源的时候由-ys 决定向每个nm申请几个cores，而一个job真正能使用到的slot是由

Flink程序运行流程分解

- [ ] 基本步骤
  - [ ] 1. Obtain an execution environment

```
getExecutionEnvironment()
createLocalEnvironment()
createRemoteEnvironment(host: String, port: Int, jarFiles: String*)
```

  - [ ] 2. Load/create the initial data

```
val text: DataStream[String] = env.readTextFile("file:///path/to/file")
```

```
```

- [ ] 3. Specify transformations on this data

```
//create a new DataStream by converting every String in the original collection
to an integer
val mapped = input.map { x => x.toInt }
```

- [ ] 4. Specify where to put the results of your computations

```
writeAsText(path: String)
print()
```

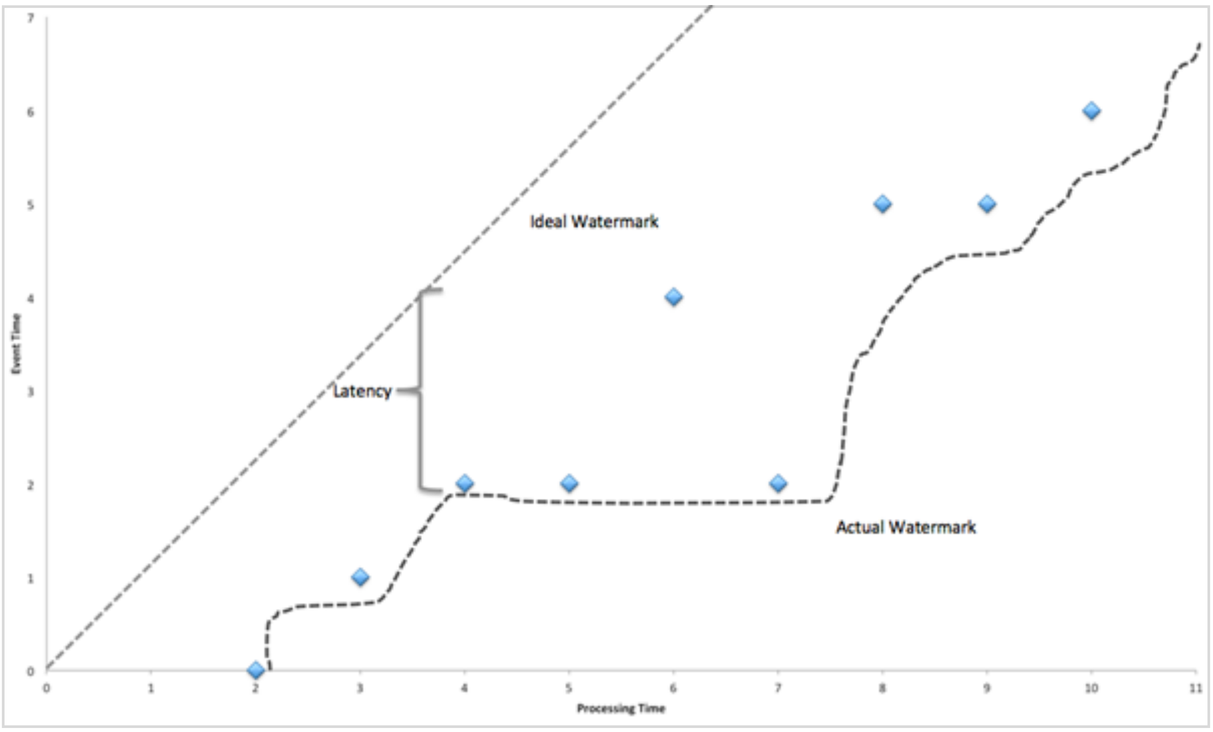
- [ ] 5. Trigger the program execution

Flink watermark机制

- [ ] 【重要】详细讲解watermark：Flink流计算编程--watermark（水位线）
    - [ ] window 触发的两个条件

```
1、watermark时间 >= window_end_time
2、在[window_start_time,window_end_time)中有数据存在
```

- [ ] 摘录：深入理解Flink核心技术
    - [ ] 纵坐标为event_time，横坐标为processingTime，理想情况自然是两者一致，但实际情况肯定不可能



- [ ] 摘录：使用EventTime与WaterMark进行流数据处理

```
 // 这块结合上图理解watermark的值
@Override
    public final Watermark getCurrentWatermark() {
        long potentialWM = currentMaxTimestamp - maxOutOfOrderness; // 当前最大
事件时间戳，减去允许最大延迟到达时间
        if (potentialWM >= lastEmittedWatermark) { // 检查上一次emit的WaterMark时
间戳，如果比lastEmittedWatermark大则更新其值
            lastEmittedWatermark = potentialWM;
        }
        return new Watermark(lastEmittedWatermark);
    }
```

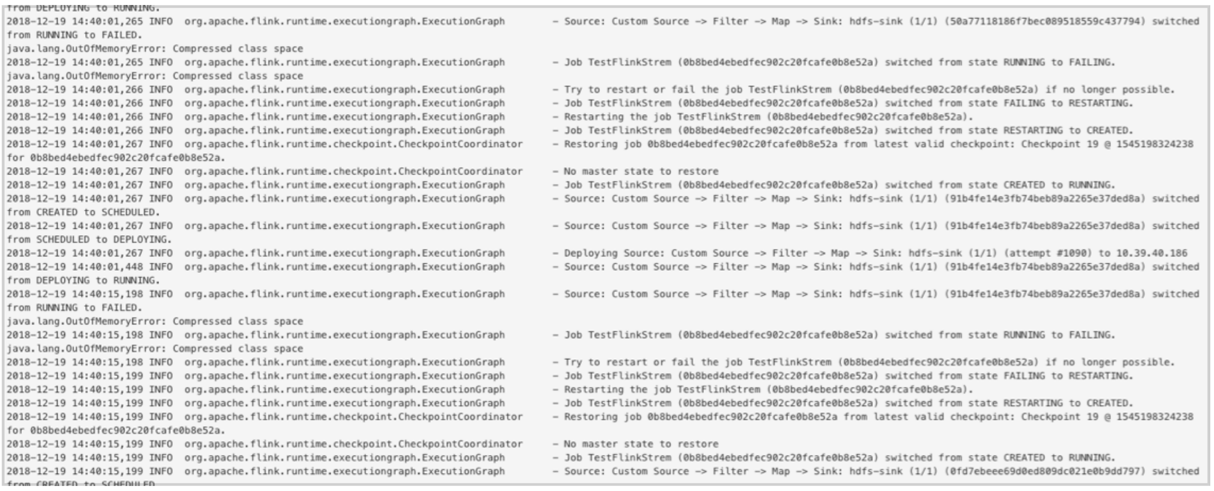
- [ ] Windowing, WaterMark,Trigger 三者依赖关系

```
1、Windowing：就是负责该如何生成Window，比如Fixed Window、Slide Window，当你配置好生成
Window的策略时，Window就会根据时间动态生成，最终得到一个一个的Window，包含一个时间范围：[起
始时间, 结束时间)，它们是一个一个受限于该时间范围的事件记录的容器，每个Window会收集一堆记录，
满足指定条件会触发Window内事件记录集合的计算处理。
2、WaterMark：它其实不太好理解，可以将它定义为一个函数E=f(P)，当前处理系统的处理时间P，根据
一定的策略f会映射到一个事件时间E，可见E在坐标系中的表现形式是一条曲线，根据f的不同曲线形状也不
同。假设，处理时间12:00:00，我希望映射到事件时间11:59:30，这时对于延迟30秒以内（事件时范围
11:59:30~12:00:00）的事件记录到达处理系统，都指派到时间范围包含处理时间12:00:00这个Window
中。事件时间超过12:00:00的就会由Trigger去做补偿了。
3、Trigger：为了满足实际不同的业务需求，对上述事件记录指派给Window未能达到实际效果，而做出的
一种补偿，比如事件记录在WaterMark时间戳之后到达事件处理系统，因为已经在对应的Window时间范围
之后，我有很多选择：选择丢弃，选择是满足延迟3秒后还是指派给该Window，选择只接受对应的Window
时间范围之后的5个事件记录，等等，这都是满足业务需要而制定的触发Window重新计算的策略，所以非常
灵活。
```

Sink Connectors

- [ ] Kafka
- [ ] Elasticsearch
- [ ] RabbitMQ
- [ ] Rolling File Sink (HDFS)
    - [ ] Apache Flink 1.7 Documentation: HDFS Connector

- [ ] Streaming File Sink
  - [ ] Apache Flink 1.7 Documentation: Streaming File Sink
  - [ ] Using Row-encoded Output Formats
    - [ ] 可以指定RollingPolicy 来滚动生成分区中的文件
  - [ ] Using Bulk-encoded Output Formats
    - [ ] 支持parquet，orc等文件格式，批量编码文件
    - [ ] 通过BulkWriter.Factory定义不同的文件格式 ParquetAvroWriters (flink 1.7-SNAPSHOT API)
    - [ ] 源码： flink/StreamingFileSink.java at master · apache/flink · GitHub
    - [ ] 使用这种方式只能配合 `OnCheckpointRollingPolicy` 使用来滚动生成分区文件，通过设置 `env.enableCheckpointing(interval)` 来设置文件滚动间隔
    - [ ] Streaming to parquet in hdfs 出现问题，内存溢出导致job无限崩溃重启，大量part file
    - [ ] partfile 有三种状态：in-progress, pending,finished；part file先被写成in-progress，一旦被关闭写入，会变成pending，当检查点成功之后，pending状态的文件将变成finished;
    - [ ] 如果失败，将从上一个检查点开始重新store，期间回滚in-progress中的文件，以确保不会重复保存上一个检查点之后的数据
    - [ ] part文件过多问题 Streaming to parquet files not happy with flink 1.6.1 - Stack Overflow
    - [ ] rolling parquet file 重点邮件 Apache Flink User Mailing List archive. - Streaming to Parquet Files in HDFS
    - [ ] 注意压缩的时候内存溢出的情况，flink陷入无限的重启循环中

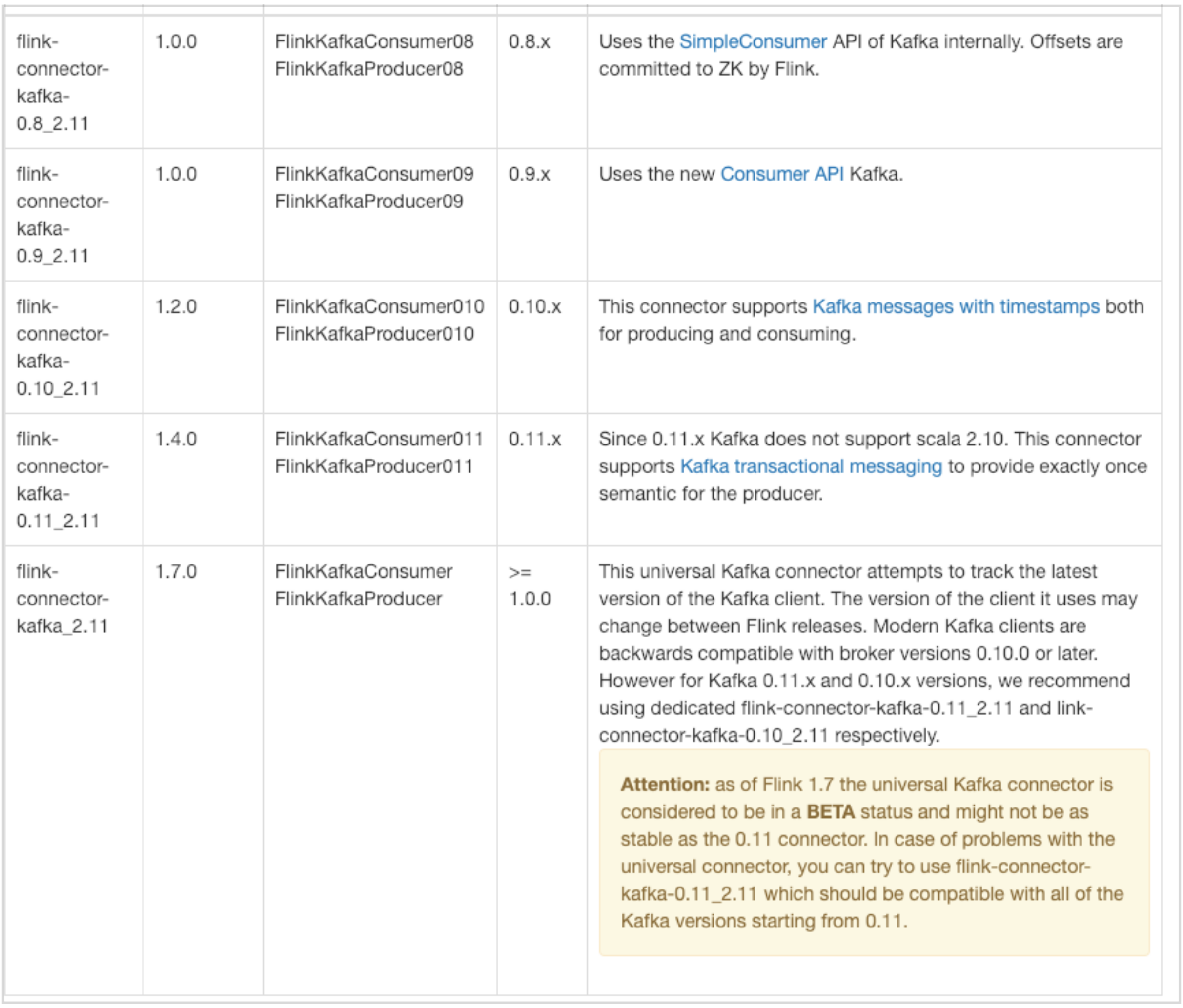


```
from DEPLOYING to RUNNING.
2018-12-19 14:40:01,265 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (50a77118186f7bec089518559c437794) switched
from RUNNING to FAILED.
java.lang.OutOfMemoryError: Compressed class space
2018-12-19 14:40:01,265 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state RUNNING to FAILING.
java.lang.OutOfMemoryError: Compressed class space
2018-12-19 14:40:01,266 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Try to restart or fail the job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) if no longer possible.
2018-12-19 14:40:01,266 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state FAILING to RESTARTING.
2018-12-19 14:40:01,266 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Restarting the job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a).
2018-12-19 14:40:01,266 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state RESTARTING to CREATED.
2018-12-19 14:40:01,267 INFO  org.apache.flink.runtime.checkpoint.CheckpointCoordinator     - Restoring job 0b8bed4ebedfec902c20fcafe0b8e52a from latest valid checkpoint: Checkpoint 19 @ 1545198324238
for 0b8bed4ebedfec902c20fcafe0b8e52a.
2018-12-19 14:40:01,267 INFO  org.apache.flink.runtime.checkpoint.CheckpointCoordinator     - No master state to restore
2018-12-19 14:40:01,267 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state CREATED to RUNNING.
2018-12-19 14:40:01,267 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (91b4fe14e3fb74beb89a2265e37ded8a) switched
from CREATED to SCHEDULED.
2018-12-19 14:40:01,267 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (91b4fe14e3fb74beb89a2265e37ded8a) switched
from SCHEDULED to DEPLOYING.
2018-12-19 14:40:01,267 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Deploying Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (attempt #1090) to 10.39.40.186
2018-12-19 14:40:01,448 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (91b4fe14e3fb74beb89a2265e37ded8a) switched
from DEPLOYING to RUNNING.
2018-12-19 14:40:15,198 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (91b4fe14e3fb74beb89a2265e37ded8a) switched
from RUNNING to FAILED.
java.lang.OutOfMemoryError: Compressed class space
2018-12-19 14:40:15,198 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state RUNNING to FAILING.
java.lang.OutOfMemoryError: Compressed class space
2018-12-19 14:40:15,198 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Try to restart or fail the job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) if no longer possible.
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state FAILING to RESTARTING.
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Restarting the job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a).
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state RESTARTING to CREATED.
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.checkpoint.CheckpointCoordinator     - Restoring job 0b8bed4ebedfec902c20fcafe0b8e52a from latest valid checkpoint: Checkpoint 19 @ 1545198324238
for 0b8bed4ebedfec902c20fcafe0b8e52a.
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.checkpoint.CheckpointCoordinator     - No master state to restore
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Job TestFlinkStrem (0b8bed4ebedfec902c20fcafe0b8e52a) switched from state CREATED to RUNNING.
2018-12-19 14:40:15,199 INFO  org.apache.flink.runtime.executiongraph.ExecutionGraph        - Source: Custom Source -> Filter -> Map -> Sink: hdfs-sink (1/1) (0fd7ebeee69d0ed809dc021e0b9dd797) switched
from CREATED to SCHEDULED.
```

flink with kafka

kafka 版本对应的功能

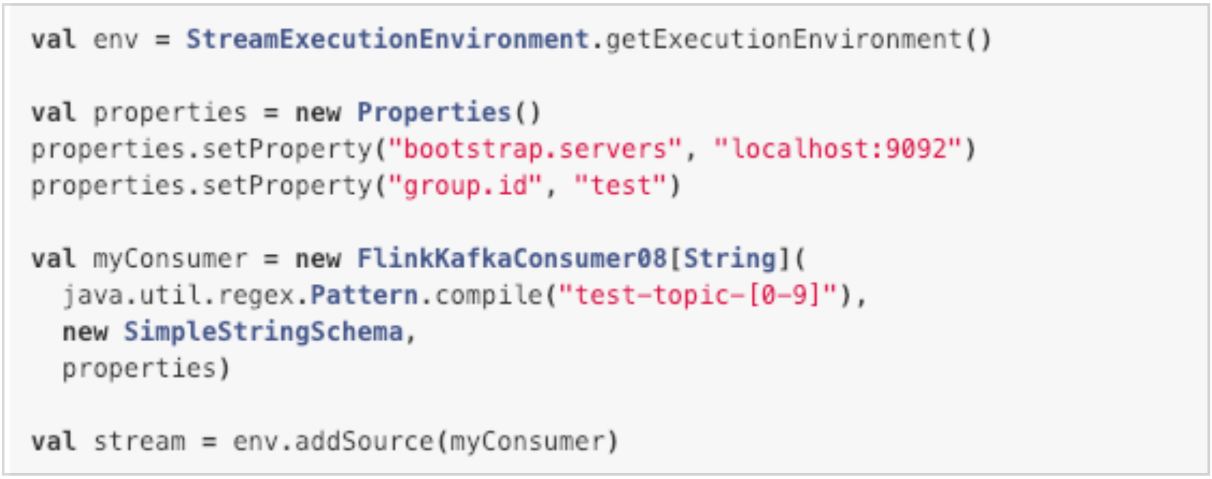
- [ ] **If checkpointing is not enabled, the Kafka consumer will periodically commit the offsets to Zookeeper.**



flink-connector-kafka-0.8_2.11	1.0.0	FlinkKafkaConsumer08 FlinkKafkaProducer08	0.8.x	Uses the SimpleConsumer API of Kafka internally. Offsets are committed to ZK by Flink.
flink-connector-kafka-0.9_2.11	1.0.0	FlinkKafkaConsumer09 FlinkKafkaProducer09	0.9.x	Uses the new Consumer API Kafka.
flink-connector-kafka-0.10_2.11	1.2.0	FlinkKafkaConsumer010 FlinkKafkaProducer010	0.10.x	This connector supports Kafka messages with timestamps both for producing and consuming.
flink-connector-kafka-0.11_2.11	1.4.0	FlinkKafkaConsumer011 FlinkKafkaProducer011	0.11.x	Since 0.11.x Kafka does not support scala 2.10. This connector supports Kafka transactional messaging to provide exactly once semantic for the producer.
flink-connector-kafka_2.11	1.7.0	FlinkKafkaConsumer FlinkKafkaProducer	>= 1.0.0	This universal Kafka connector attempts to track the latest version of the Kafka client. The version of the client it uses may change between Flink releases. Modern Kafka clients are backwards compatible with broker versions 0.10.0 or later. However for Kafka 0.11.x and 0.10.x versions, we recommend using dedicated flink-connector-kafka-0.11_2.11 and link-connector-kafka-0.10_2.11 respectively. **Attention:** as of Flink 1.7 the universal Kafka connector is considered to be in a **BETA** status and might not be as stable as the 0.11 connector. In case of problems with the universal connector, you can try to use flink-connector-kafka-0.11_2.11 which should be compatible with all of the Kafka versions starting from 0.11.

开启自动发现机制

- [ ] 分区自动发现
  - [ ] By default, partition discovery is **disabled.** To enable it, set a non-negative value for **flink.partition-discovery.interval-millis** in the provided properties config, representing the discovery interval in **milliseconds**.
- [ ] topic 自动发现
  - [ ] based on pattern matching on the topic names using regular expressions
  - [ ] 同样需要设置**flink.partition-discovery.interval-millis**



```
val env = StreamExecutionEnvironment.getExecutionEnvironment()

val properties = new Properties()
properties.setProperty("bootstrap.servers", "localhost:9092")
properties.setProperty("group.id", "test")

val myConsumer = new FlinkKafkaConsumer08[String](
  java.util.regex.Pattern.compile("test-topic-[0-9]"),
  new SimpleStringSchema,
  properties)

val stream = env.addSource(myConsumer)
```

produce message to kafka with flink

- [ ] docApache Flink 1.7 Documentation: Apache Kafka Connector

```
val stream: DataStream[String] = ...

val myProducer = new FlinkKafkaProducer011[String](
        "localhost:9092",         // broker list
        "my-topic",               // target topic
        new SimpleStringSchema)   // serialization schema

// versions 0.10+ allow attaching the records' event timestamp when writing them to Kafka;
// this method is not available for earlier Kafka versions
myProducer.setWriteTimestampToKafka(true)

stream.addSink(myProducer)
```

- [ ] 如果不指定partitioner的话，默认是FlinkFixedPartitioner
  - [ ] maps each Flink Kafka Producer parallel subtask to a single Kafka partition (i.e., all records received by a sink subtask will end up in the same Kafka partition
- [ ] Kafka 0.9 and 0.10
  - [ ] setLogFailureOnly=false, setFlushOnCheckpoint=true provide **at-least-once** guarantees
  - [ ] **默认retries 为0**，是为了防止重试导致消息重复
- [ ] Kafka 0.11 and newer
  - [ ] provide **exactly-once delivery guarantees**
  - [ ] use FlinkKafkaProducer011 with semantic settings
    - [ ] Semantic.NONE
    - [ ] Semantic.AT_LEAST_ONCE
    - [ ] Semantic.EXACTLY_ONCE
      - [ ] 同样得设置事务隔离级别，read_committed，read_uncommitted 默认是 read_uncommitted

StreamingFileSink与Kafka 结合如何做到exactly once?

- [ ] An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!)
  - [ ] 二阶段提交
- [ ] partfile 有三种状态：in-progress, pending,finished；part file先被写成in-progress，一旦被关闭写入，会变成pending，当检查点成功之后，pending状态的文件将变成finished;
- [ ] 如果失败，将从上一个检查点开始重新store，期间回滚in-progress中的文件，以确保不会重复保存上一个检查点之后的数据

flink如何控制kafka offset提交与checkpoint&&savepoint相结合

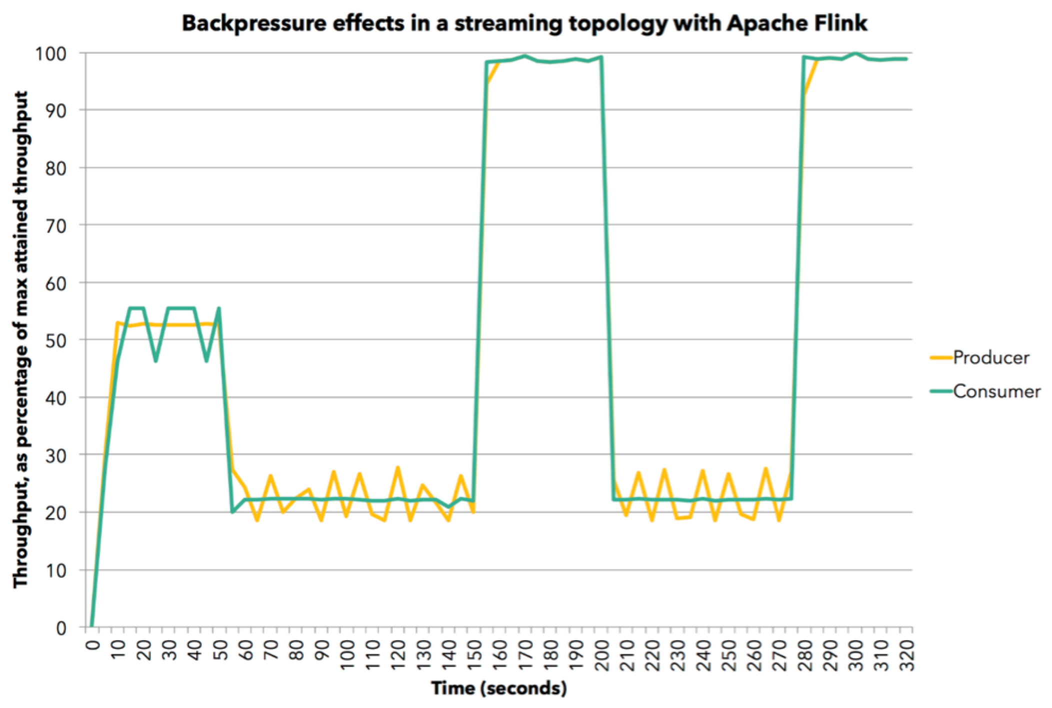
- [ ] FlinkKafkaConsumer使用详解
- [ ] Flink 小贴士 (2)：Flink 如何管理 Kafka 消费位点 | Jark's Blog
- [ ] 关闭checkpoint(Checkpointingdisabled):
  - [ ] 此时， Flink Kafka Consumer依赖于它使用的具体的Kafka client的自动定期提交offset的行为，相应的设置是 Kafka properties中的 enable.auto.commit (或者 auto.commit.enable 对于Kafka 0.8) 以及 auto.commit.interval.ms

- [ ] 开启checkpoint(Checkpointingenabled):
    - [ ] 在这种情况下，Flink Kafka Consumer会将offset存到checkpoint中
    - [ ] 制作完checkpoint 一并提交offsets 当checkpoint 处于completed的状态时（整个job的所有的operator都收到了这个checkpoint的barrier）。将offset记录起来并提交，从而保证exactly-once
- [ ] exactly once的两个风险点：可结合savepoint来做
    - [ ] 1. 异常退出的情况，没法来得及做checkpoint，而checkpoint间隔太长会导致丢失大量数据；可以通过airflow周期性手动触发savepoint恢复；封装hflink脚本
        - [ ] 解决思路是结合savepoint来做，通过airflow定时的触发savepoint操作，防止因checkpoint未及时做数据丢失
        - [ ] 规定一分钟savepoint一次，这样即使分钟级别的数据丢失也是可以容忍
    - [ ] 2. 第一点利用savepoint来做也有风险：在做savepoint的时候，如果异常退出，parfile未及时关闭导致数据丢失
        - [ ] 暂时可以认为问题较小？

flink如何控制背压

- [ ] 如何做到挂很久之后重新启动时限制拉取的消息量？（类似spark max.perpartition）
    - [ ] 背压通过task slot 的stackTrace判断
    - [ ] 可以在kafka source那层控制一次性消费量，类似于spark
    - [ ] 其实flink天然就是背压的状态
        - [ ] 不用担心一次性拉取全部的量，有个默认的max.partition.fetch.bytes=1048576；可以背压从sink到source 的整个pipeline,同时对source进行限流来适配整个pipeline中最慢组件的速度，从而获得系统的稳定状态。
        - [ ] 也不用担心一次拉取的量不够，通过背压可以调整

吐。



首先，我们运行生产者task到它最大生产速度的60%（我们通过Thread.sleep()来模拟降速）。消费者以同样的速度处理数据。 然后，我们将消费task的速度降至其最高速度的30%。你就会看到背压问题产生了，正如我们所见，生产者的速度也自然降至其最高速度的30%。 接着，我们对消费者停止人为降速，之后生产者和消费者task都达到了其最大的吞吐。接下来，我们再次将消费者的速度降至30%，pipeline给出了立即响应：生产者的速度也被自动降至30%。 最后，我们再次停止限速，两个task也再次恢复100%的速度。这所有的迹象表明：生产者和消费者在pipeline中的处理都在跟随彼此的吞吐而进行适当的调整，这就是我们在流pipeline中描述的行为。

Flink 如何管理 Kafka 消费位点

- [ ] Flink 小贴士 (2)：Flink 如何管理 Kafka 消费位点 | Jark's Blog

Flink 高性能部署

- [ ] Apache Flink 1.8-SNAPSHOT Documentation: JobManager High Availability (HA)

Flink 内存管理

- [ ] 001待整理：flink的背压机制 · flink简明实战教程
- [ ] Flink 原理与实现：内存管理 - 简书
- [ ] Flink序列化框架分析_慕课手记
- [ ] Flink 原理与实现：内存管理 | Jark's Blog
  - [ ] **有效避免OOM**所有的运行时数据结构和算法只能通过**内存池**申请内存，保证了其使用的内存大小是固定的，不会因为运行时数据结构和算法而发生OOM。在内存吃紧的情况下，算法（sort/join等）会高效地将一大批内存块写到磁盘，之后再读回来。因此，OutOfMemoryErrors可以有效地被避免
  - [ ] 堆外内存在写磁盘或网络传输时是 zero-copy，而堆内存的话，至少需要 copy 一次

Flink Broadcast State

- [ ] Flink 小贴士 (6): 使用 Broadcast State 的 4 个注意事项 | Jark's Blog
- [ ] A Practical Guide to Broadcast State in Apache Flink
- [ ] Flink Broadcast 广播变量应用案例实战-Flink牛刀小试 - 掘金

相关问题

- [ ] 不允许null field或instance在operator间传递，即使代码写了filter；这一点跟spark处理机制不同Re: Cannot pass objects with null-valued fields to the next operator

metric监控rest api

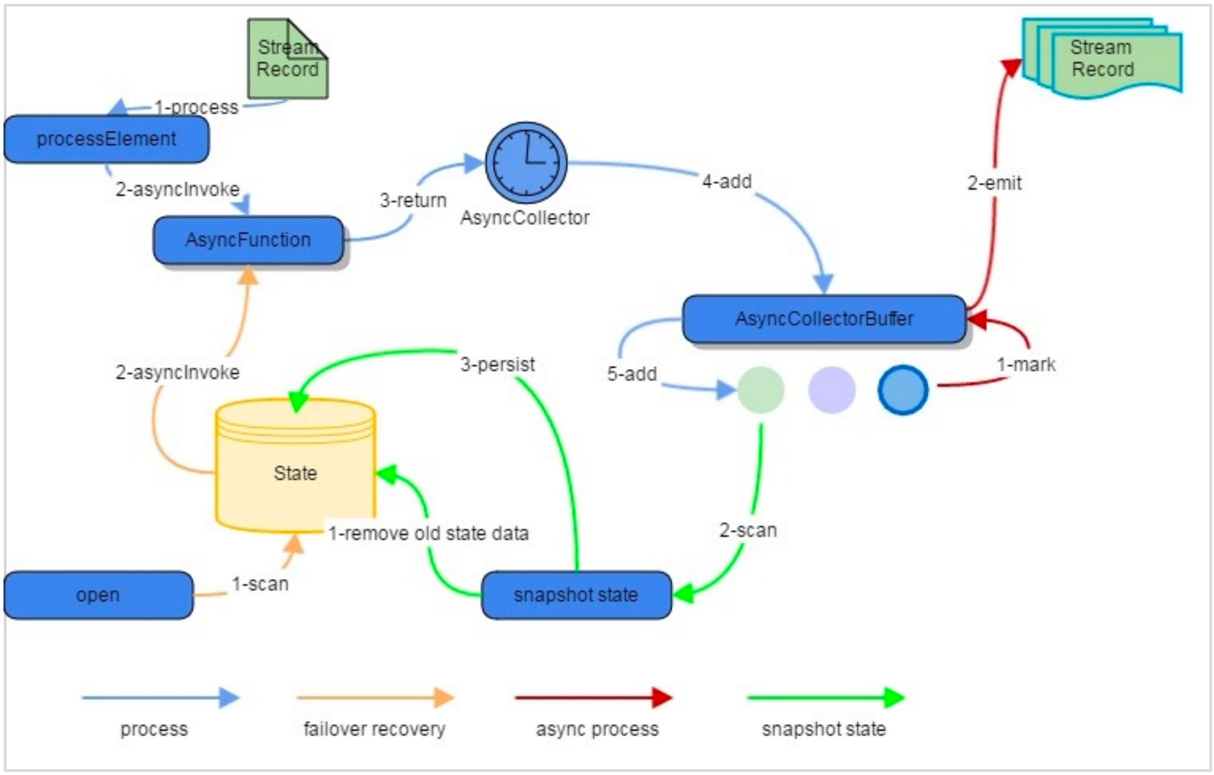
- [ ] Apache Flink 1.8-SNAPSHOT Documentation: Monitoring REST API
- [ ] Flink 监控上游Kafka的情况
    - [ ] 从 Kafka version 0.9开始，kafka consumer 把所有的metric暴露了出来，所有已经暴露的指标见文档 The Kafka documentation lists all exported metrics
    - [ ] fetch-size-avg: The average number of bytes fetched per request for a topic
    - [ ]

Restart Strategies

- [ ] doc Apache Flink 1.7 Documentation: Restart Strategies
- [ ] Fixed Delay Restart Strategy
- [ ] Failure Rate Restart Strategy
- [ ] No Restart Strategy
- [ ] Fallback Restart Strategy

Flink with async IO

- [ ] Apache Flink 1.7 Documentation: Asynchronous I/O for External Data Access
- [ ] 设计思想：FLIP-12: Asynchronous I/O Design and Implementation - Apache Flink - Apache Software Foundation



Flink with table api

- [ ] Apache Flink 1.7 Documentation: Concepts & Common API
- [ ] [Table Api操作符 sum, max...](Apache Flink 1.7 Documentation: Built-In Functions)

Flink for Cloudera

- [ ] flink 默认不支持cdh版本Hadoop，需要重新编译一下flink源码
    - [ ] https://ci.apache.org/projects/flink/flink-docs-release-1.7/flinkDev/building.html

```
mvn clean install -DskipTests -Pvendor-repos -Dhadoop.version=3.0.0-cdh6.0.1
```

    - [ ] 配置maven仓库 解决maven仓库默认不支持cdh - 这里的天空比较蓝 - CSDN博客

```
<mirror>
    <id>nexus-aliyun</id>
    <mirrorOf>*,!cloudera</mirrorOf>
    <name>Nexus aliyun</name>
    <url>
      http://maven.aliyun.com/nexus/content/groups/public
    </url>
</mirror>
```