



Spark Streaming

董炫辰

IBM Platform Computing

xcdong@cn.ibm.com

Agenda

- Spark Streaming介绍
- Storm/Flink介绍
- Spark Streaming开发接口
- Flume/Kafka接入
- 开发实例



概述:



概述:

```
val ssc = new StreamingContext(sparkConf, Seconds(2))
val lines = ssc.textFileStream("xxxx")
    lines: org.apache.spark.streaming.dstream.DStream[String] =
    org.apache.spark.streaming.dstream.MappedDStream@48342983
val words = lines.flatMap(_.split(" "))
    words: org.apache.spark.streaming.dstream.DStream[String] =
    org.apache.spark.streaming.dstream.FlatMappedDStream@1f1eb3f3
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts: org.apache.spark.streaming.dstream.DStream[(String, Int)]
    = org.apache.spark.streaming.dstream.ShuffledDStream@2301c03f
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

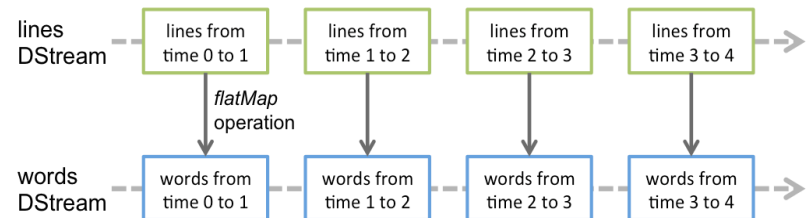
Dstream:

Dstream is the basic abstraction provided by Spark Streaming.

a DStream is represented by a continuous series of RDDs

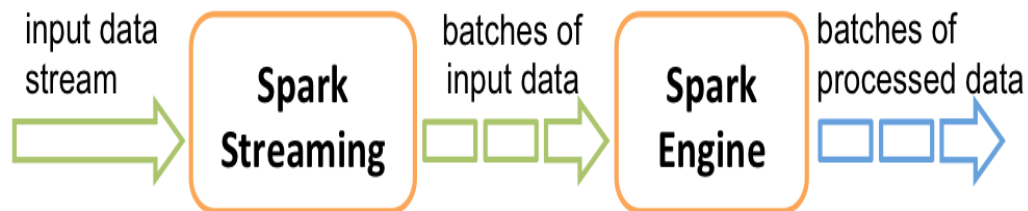


Any operation applied on a DStream translates to operations on the underlying RDDs.



These underlying RDD transformations are computed by the Spark engine.

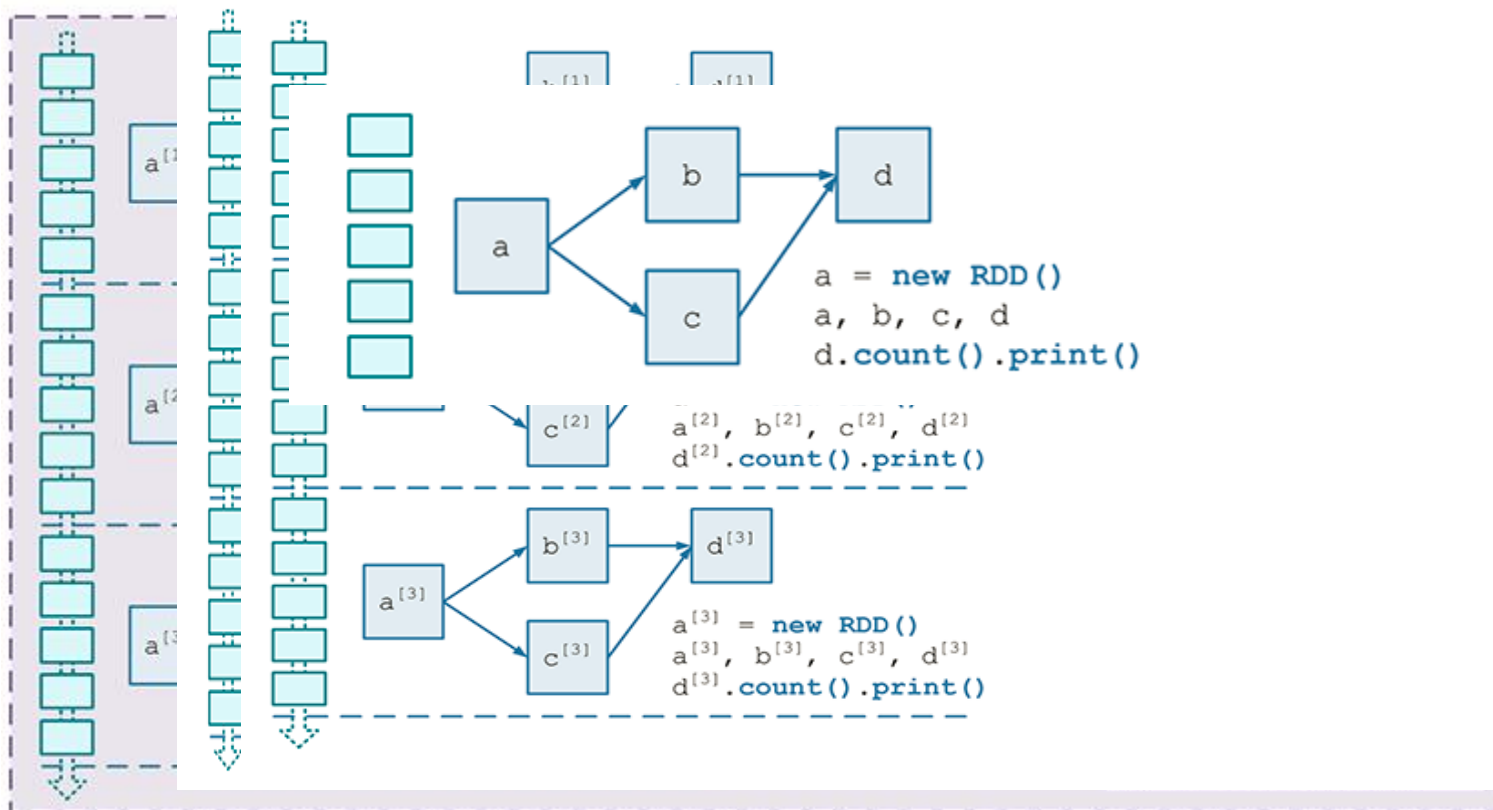
原理：



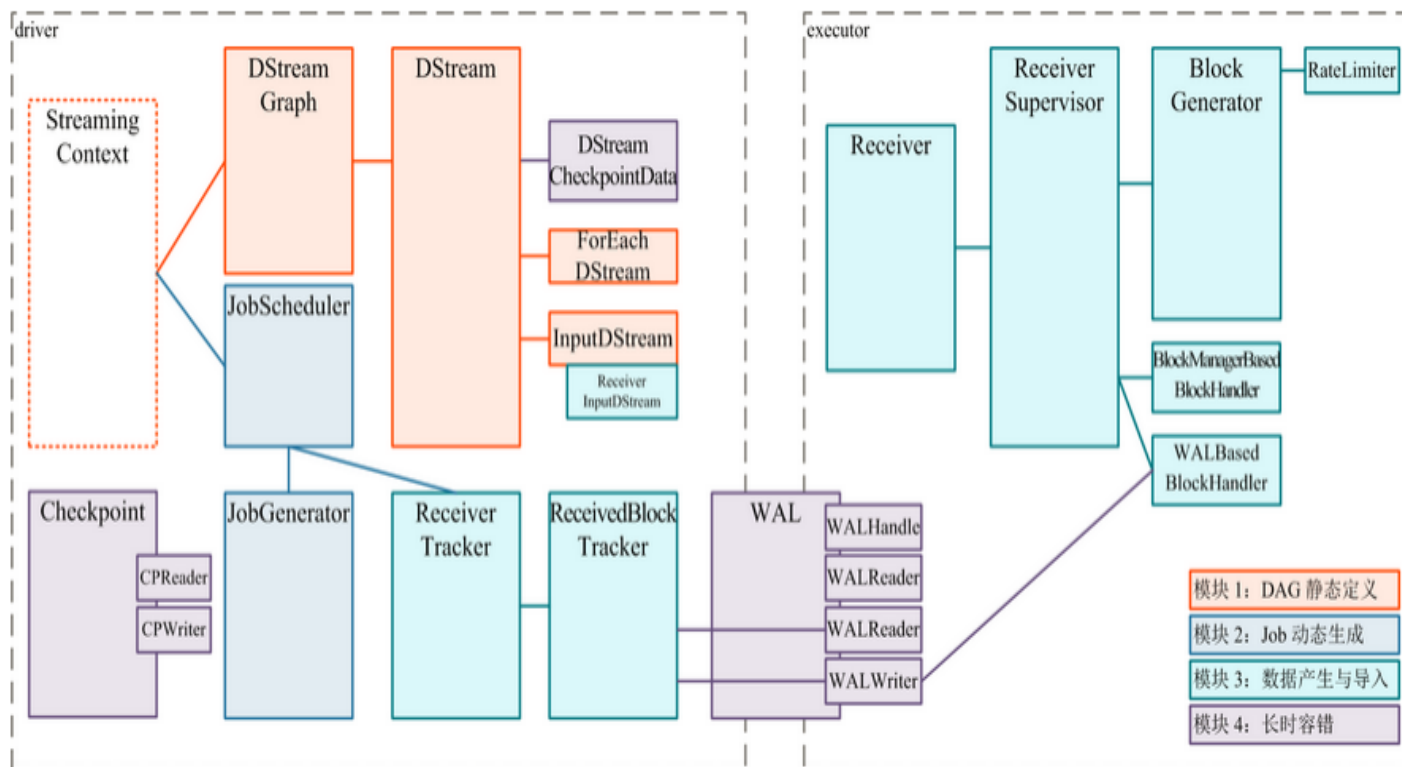
sparkstreaming按照Dstream对数据进行各种变换，其内部是将数据流以时间片（秒级）为单位进行拆分，然后以类似批处理的方式处理每个时间片数据。

每个时间片数据就是一个mini batch的RDD，处理方式和之前spark core处理RDD的方式一样

需要解决的问题：

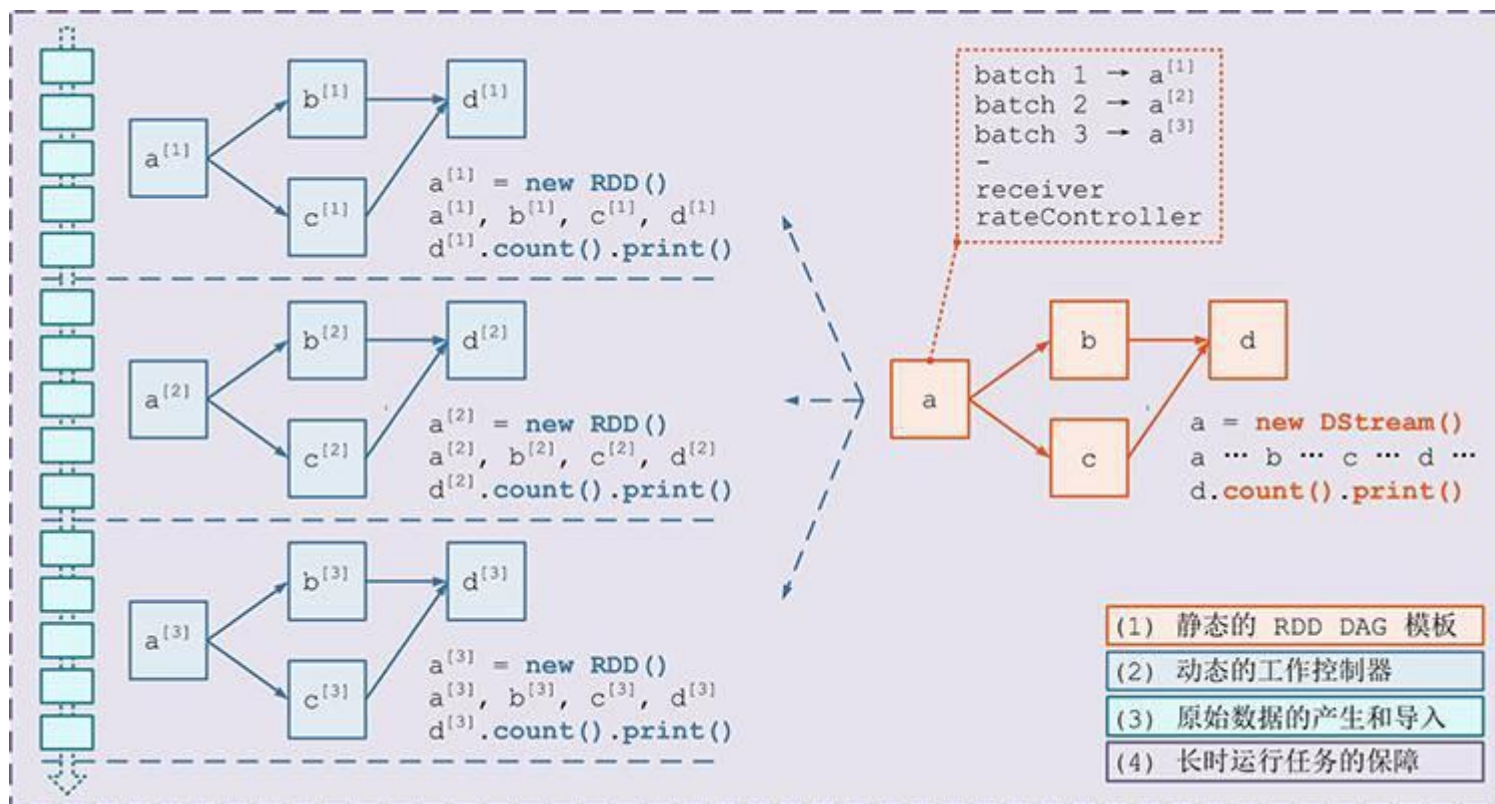


整体模块划分

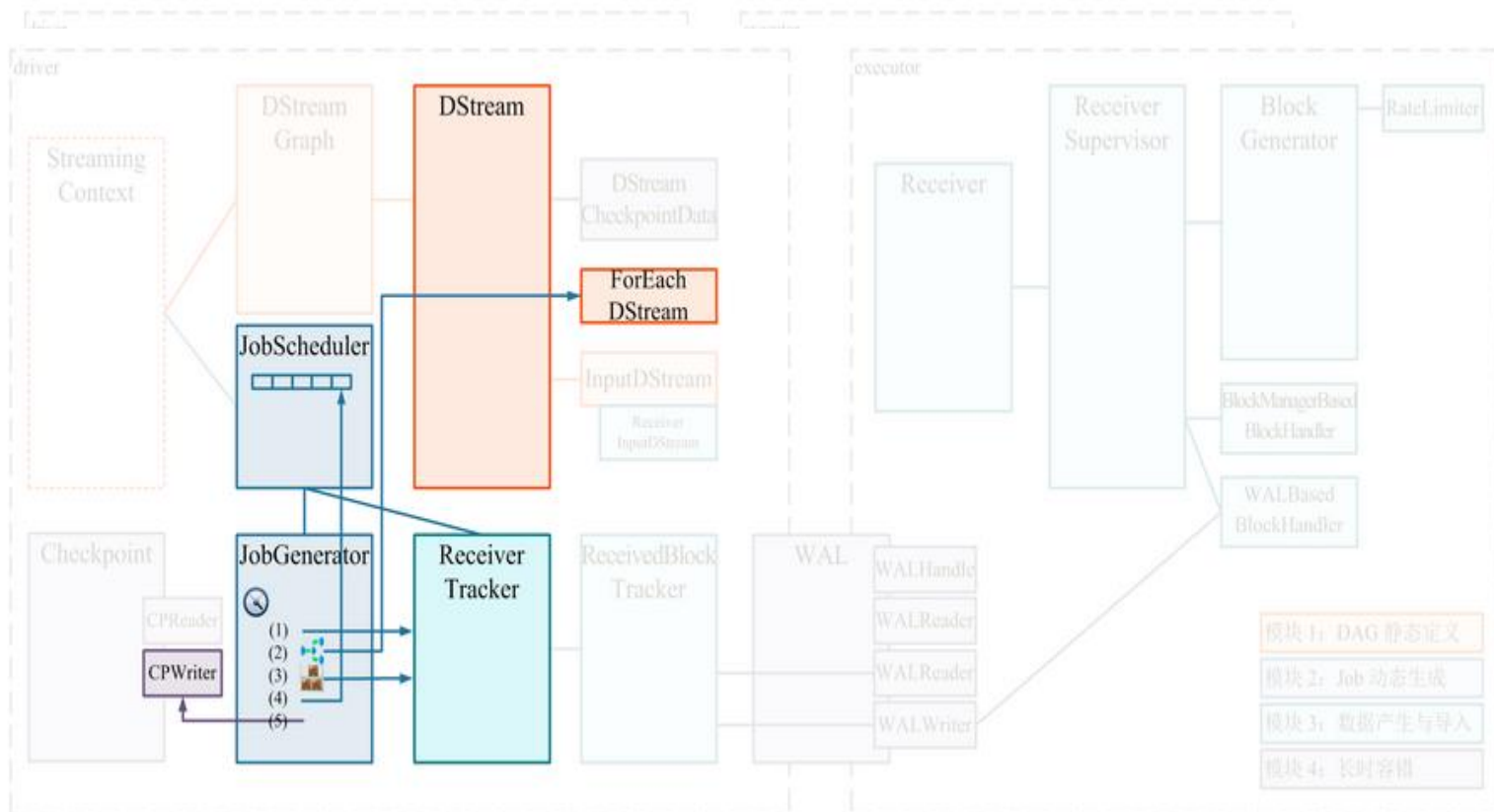


模块 1: DAG 静态定义

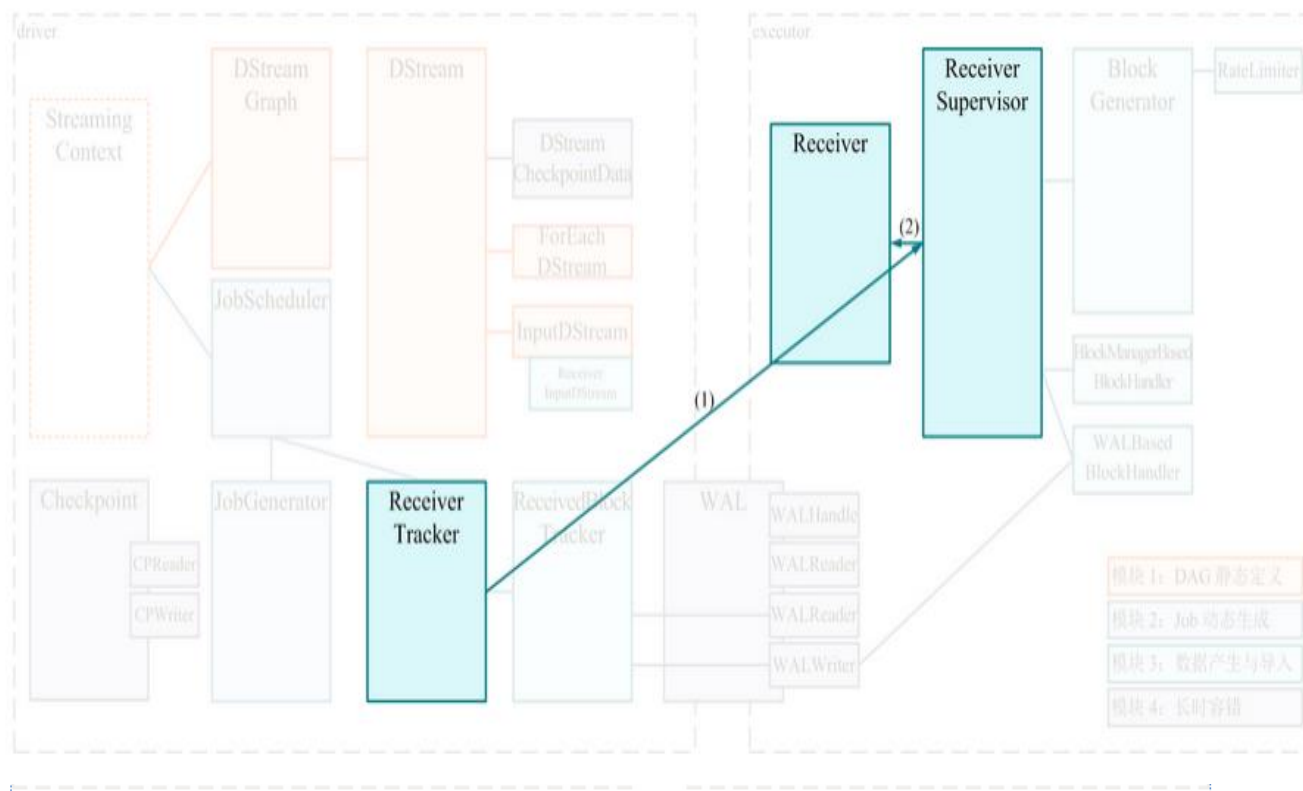
RDD, DStream 和 DStreamGraph



模块 2: Job 动态生成



模块 3：数据产生与导入



模块4：长时容错

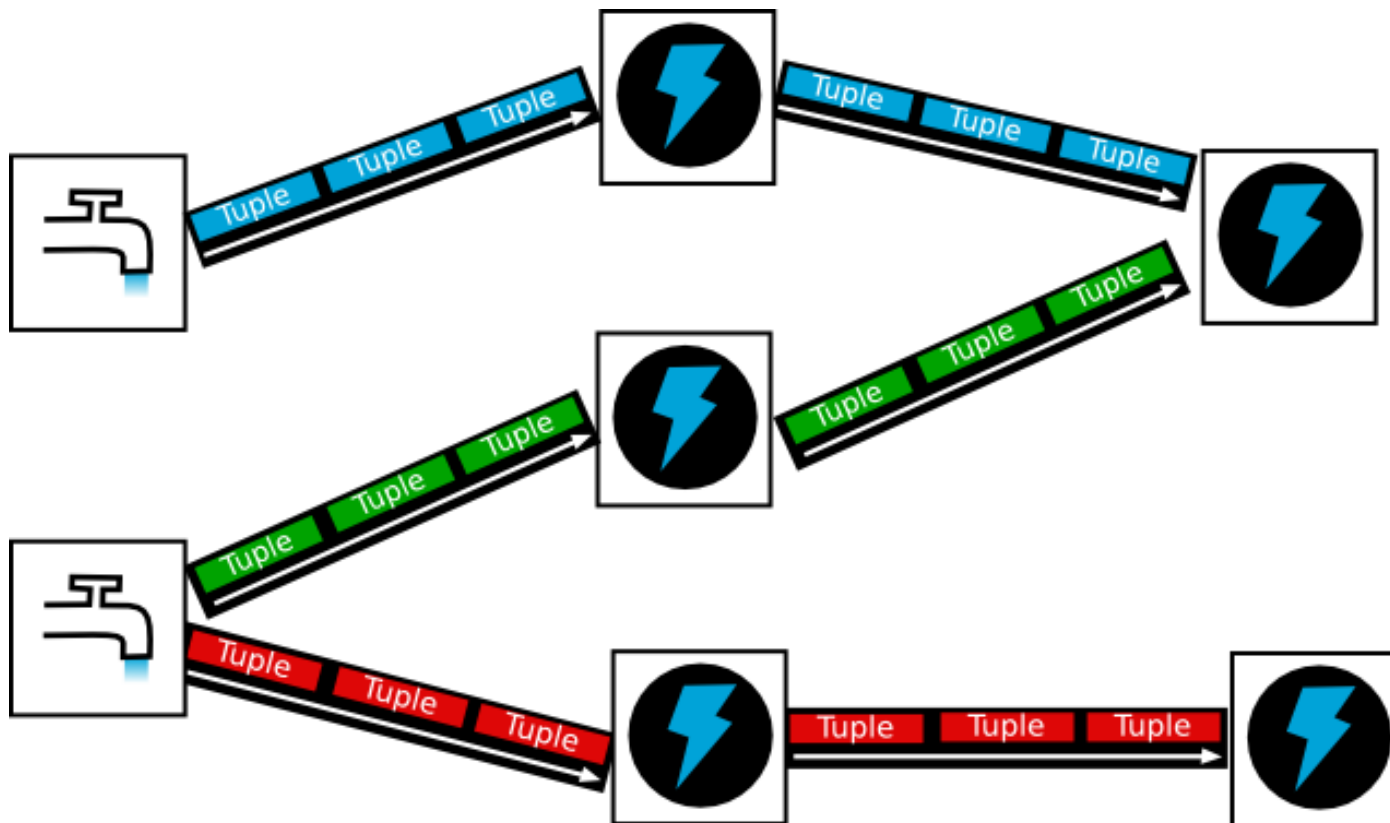
1.executor端的容错

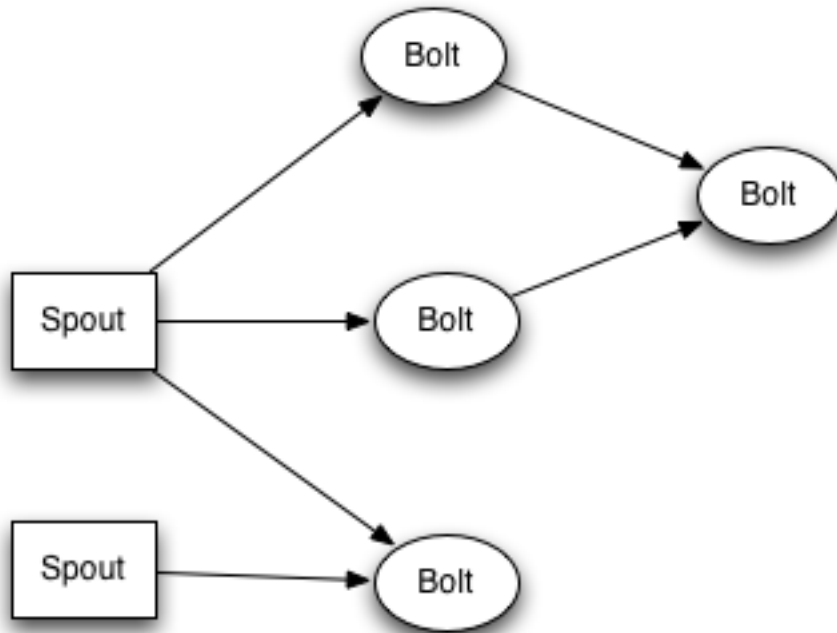
	图示	优点	缺点
(1) 热备		无 recover time	需要占用双倍资源
(2) 冷备		十分可靠	存在 recover time
(3) 重放		不占用额外资源	存在 recover time
(4) 忽略		无 recover time	准确性有损失

2.driver端的容错

- a.ReceivedBlockTracker (meta信息)也采用 WAL 冷备方式进行备份
- b.Checkpoint, 来记录整个 DStreamGraph 的变化、和每个 batch 的 job 的完成情况（两次，提交后和job完成后）。

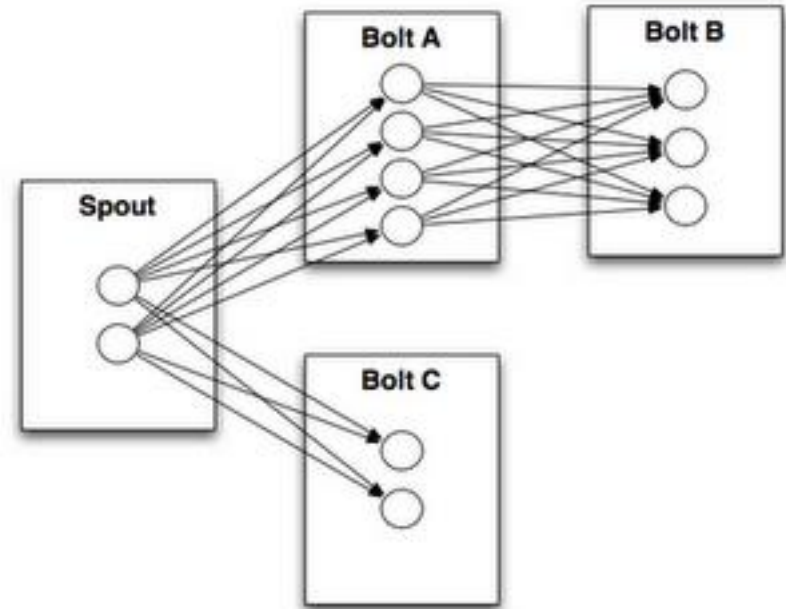
Storm使用场景：实时流数据处理





主要概念有:

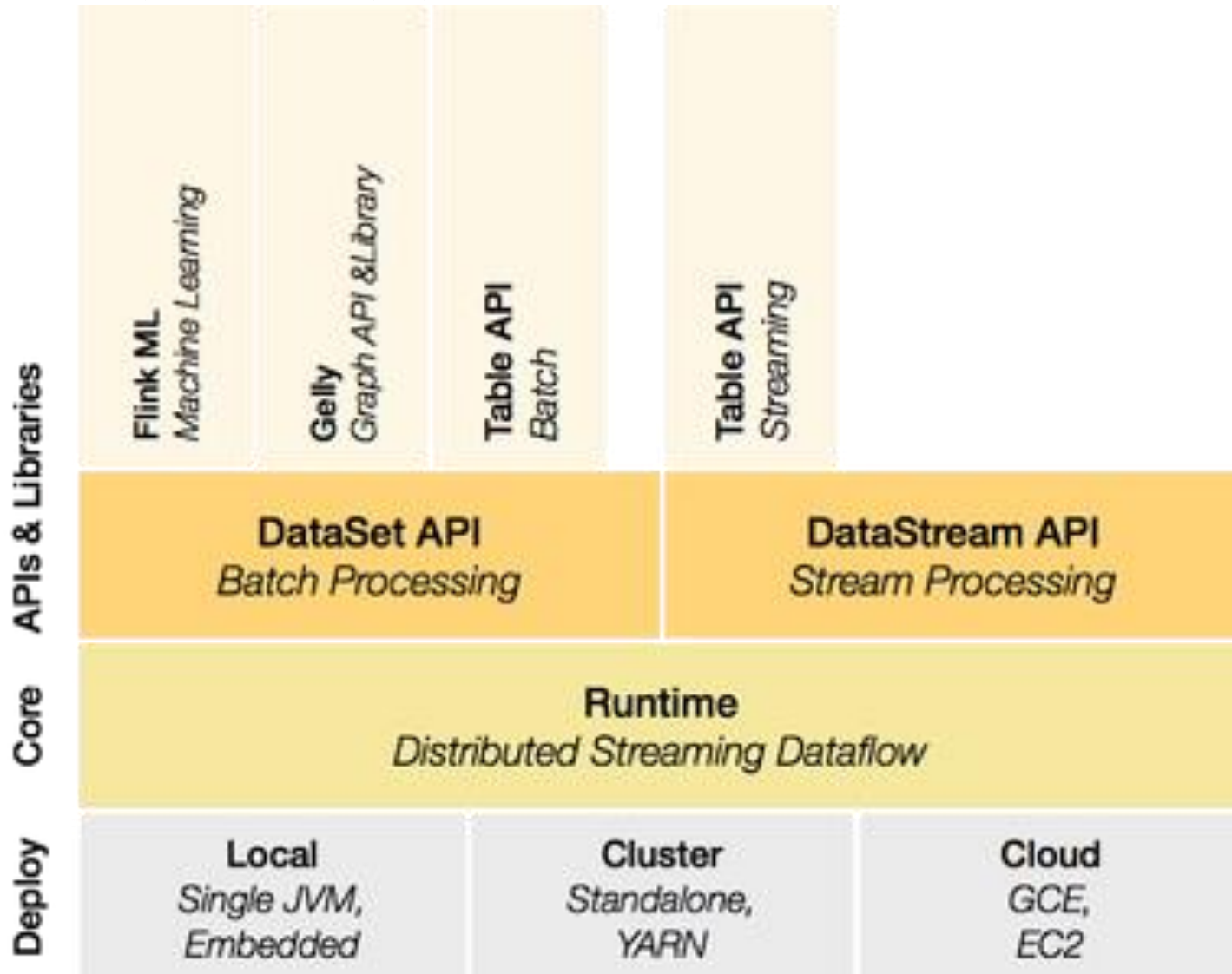
- 拓扑 (Topologies)
- 元组 (Tuple)
- Spouts (喷嘴)
- Bolts
- 流分组 (Stream groupings)

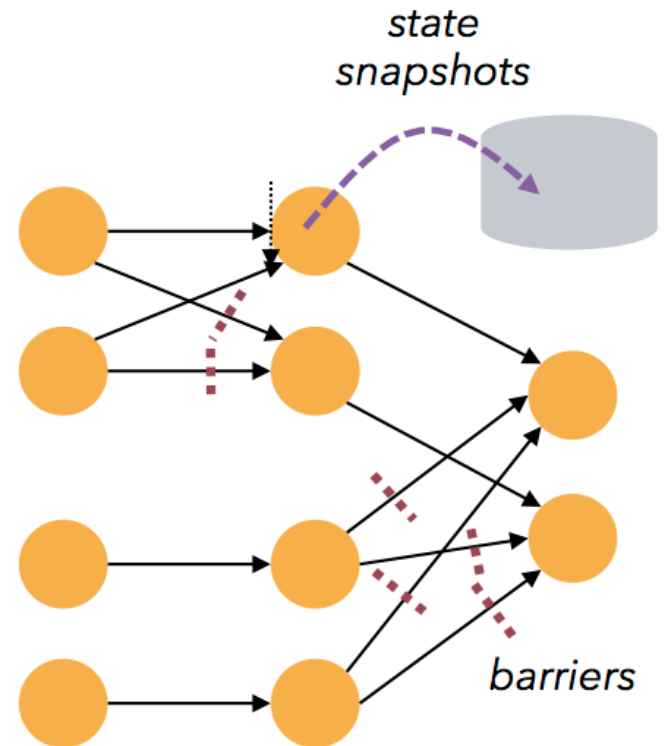
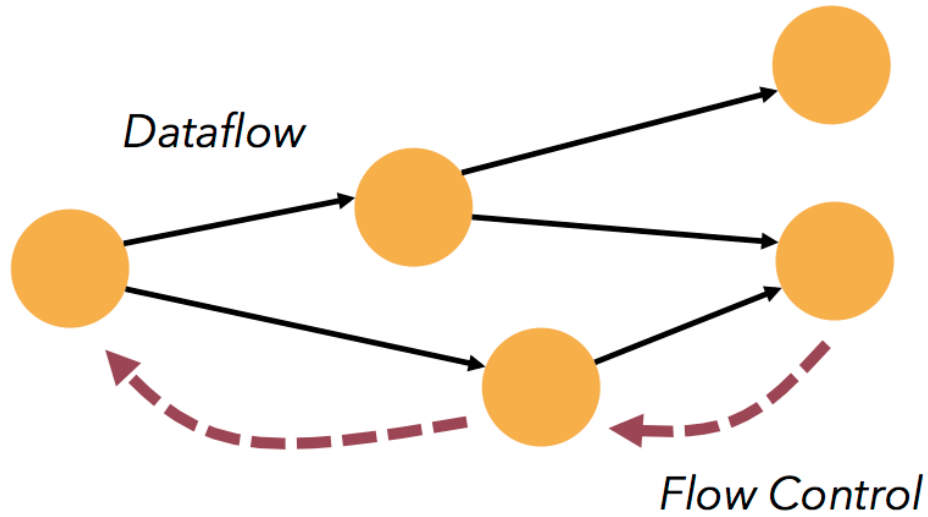


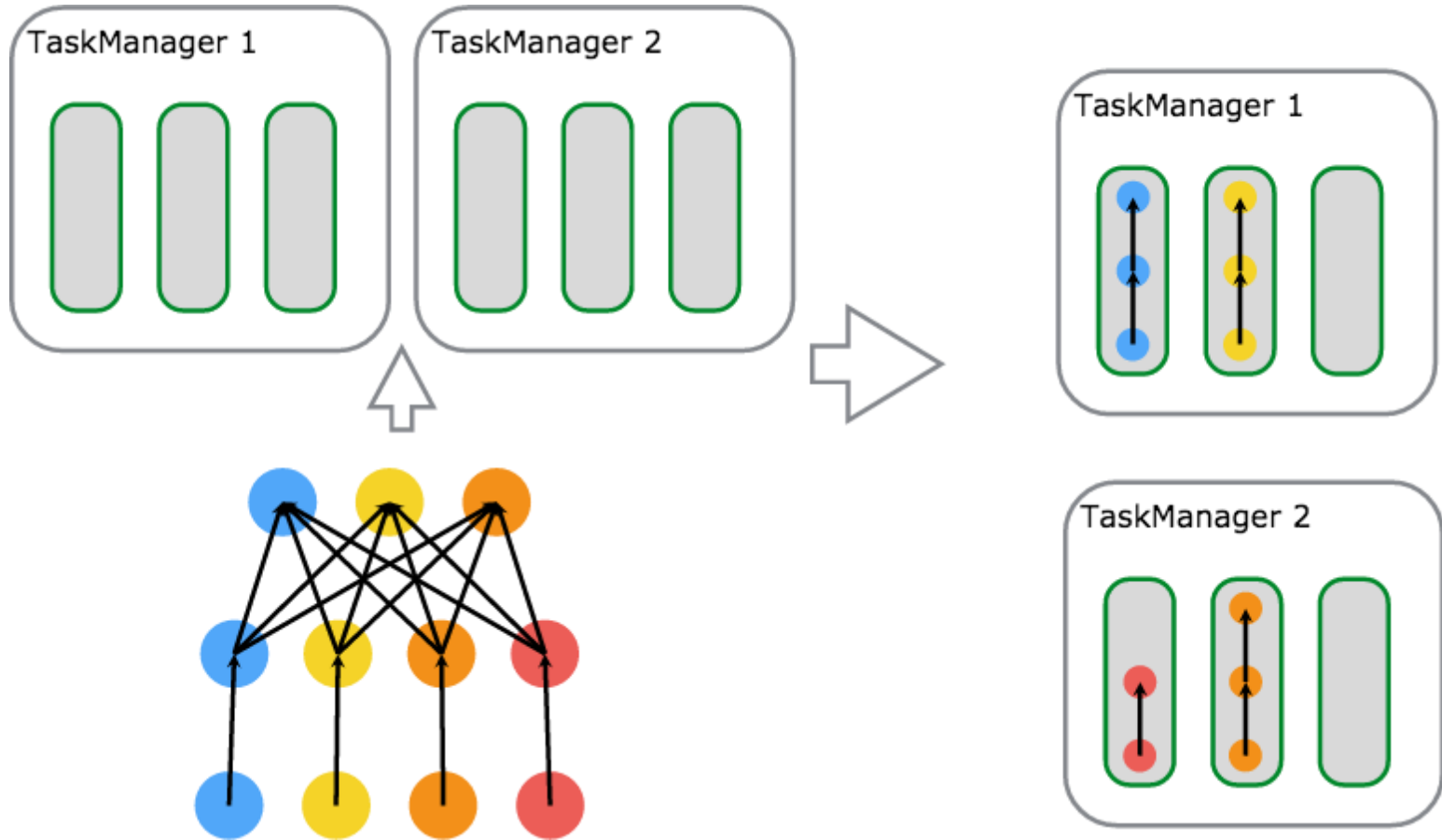
流分组:

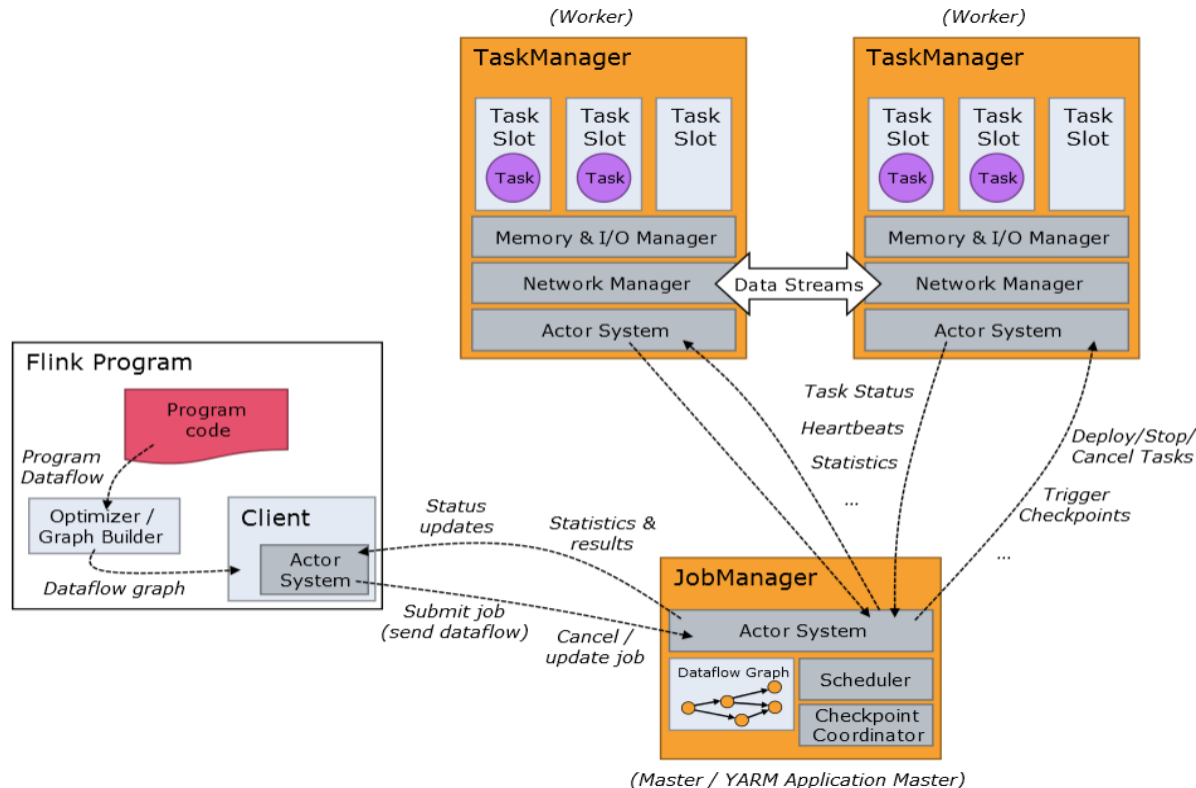
- 洗牌分组 (Shuffle grouping)
- 字段分组 (Fields grouping)
- Partial Key grouping
- All grouping
- Global grouping
- 不分组 (None grouping)
- Direct grouping
- Local or shuffle grouping

Flink使用场景：实时流数据处理，批处理





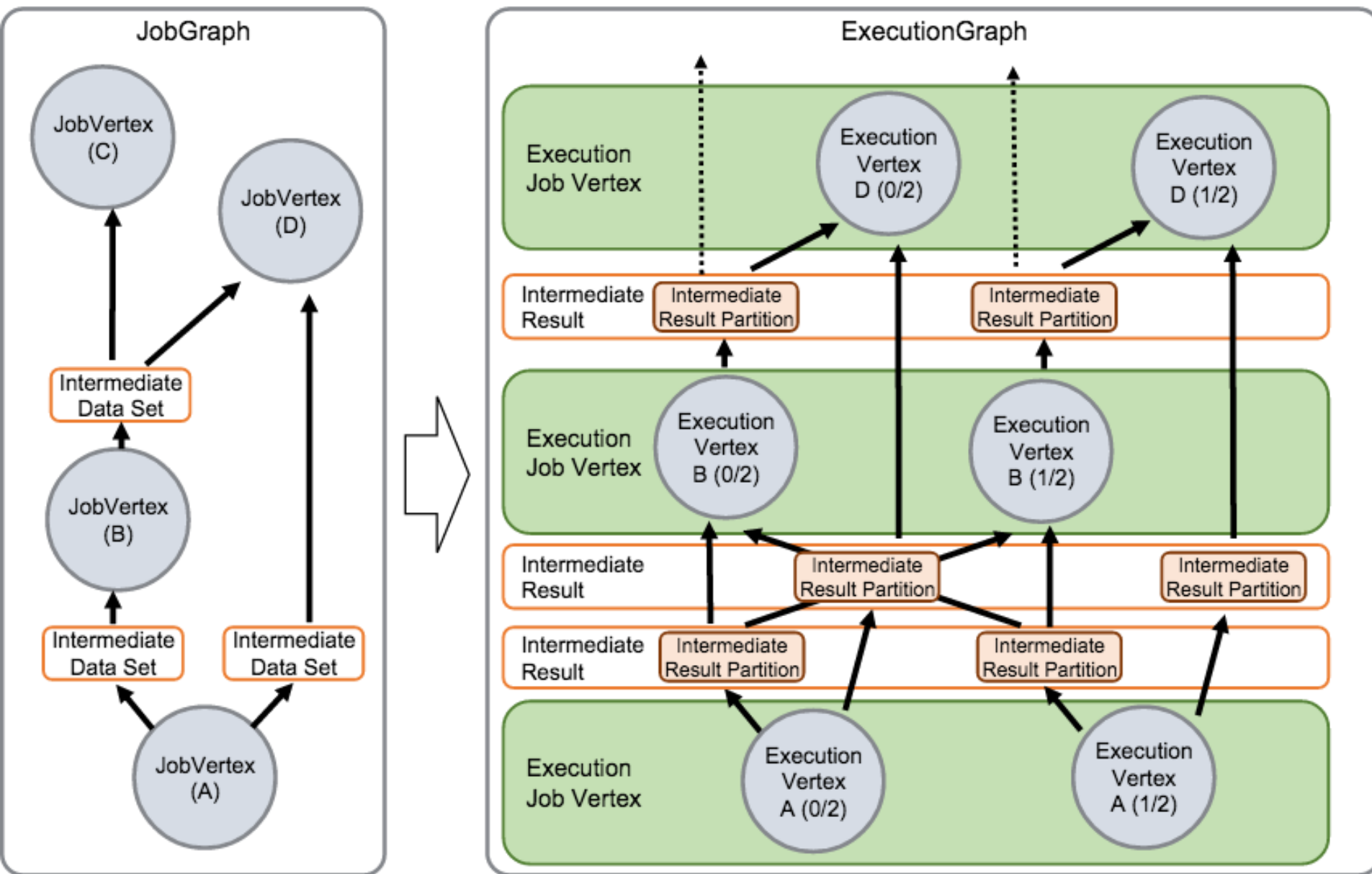




```

case class Word(word: String, freq: Long)
val texts: DataStream[String] = ...
val counts = text
    .flatMap { line => line.split("\\W+") }
    .map { token => Word(token, 1) }
    .keyBy("word")
    .timeWindow(Time.seconds(5), Time.seconds(1))
    .sum("freq")

```



优势:

- 统一的开发接口
- 吞吐和容错
- 多种开发范式混用, Streaming + SQL, Streaming + MLlib
- 利用Spark内存pipeline计算(?)
- 社区活跃: spark(820), flink(156), storm(200)

劣势:

- 微批处理模式, 准实时(s级别)

1. Define the input sources by **creating input DStreams**.
2. Define the streaming computations by applying **transformation and output operations** to DStreams.
3. Start **receiving data** and processing it using `streamingContext.start()`.
4. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`.
5. The processing can be manually stopped using `streamingContext.stop()`.

■ 初始化StreamingContext:

```
import org.apache.spark._  
import org.apache.spark.streaming._  
val conf = new SparkConf().setAppName(appName).setMaster(master)  
val ssc = new StreamingContext(conf, Seconds(1))
```

或者

```
val sc = ... // existing SparkContext  
val ssc = new StreamingContext(sc, Seconds(1))
```

■ 注意:

1. 一次只能有一个streamingcontext, 并且一旦被停止, 就不能被重启
2. 使用ssc.stop()会把内部关联的sparkcontext也停掉。可以使用
ssc.stop(stopSparkContext = False)来避免这样做
3. sparkcontext可以被重用, 创建多个streamingcontext, 但是要用新的streamingcontext, 必须把原来alive的streamingcontext停掉

■ Input DStreams and Receivers:

- Basic sources: Sources directly available in the StreamingContext API.
Examples: **file systems**, **socket connections**, and **Akka actors**.
- Advanced sources: Sources like **Kafka**, **Flume**, **Kinesis**, **Twitter**, **MQTT**, **Zeromq**, etc. are available through extra utility classes.

Kinesis: 亚马逊提供的服务，它支持开发者从不同来源传输大量的数据并进行处理

MQTT: IBM开发的即时通讯协议，有可能成为物联网的重要组成部分。该协议几乎可以把所有联网物品 和外部连接起来，被用来当做传感器和致动器（比如通过Twitter让房屋联网）的通信协议。

Zeromq: 类似于Socket的一系列接口，他跟Socket的区别是：普通的socket是端到端的（1:1的关系），而ZMQ却是可以N: M 的关系。ZMQ用于node与node间的通信，node可以是主机或者是进程。

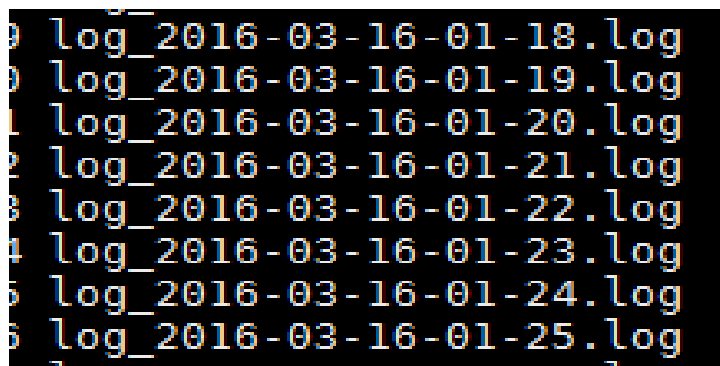
举例：

```
ssc.socketTextStream("localhost", 9999)
ssc.textFileStream("xxx")
ssc.actorStream[String](
  Props(new SampleActorReceiver[String]), "SampleReceiver", storageLevel, supervisorStrategy)
TwitterUtils.createStream(ssc, None)
KafkaUtils.createStream(ssc, zkQuorum, group, topicMap)
```

■ 插播:

log4j按时间滚动日志配置

```
log4j.rootCategory=info, stdout, logfile
log4j.appender.logfile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.logfile.File= ./log/log_
log4j.appender.logfile.DatePattern=yyyy-MM-dd-HH-mm'.log'
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] -
<%m>%n
```



```
0 log_2016-03-16-01-18.log
0 log_2016-03-16-01-19.log
1 log_2016-03-16-01-20.log
2 log_2016-03-16-01-21.log
3 log_2016-03-16-01-22.log
4 log_2016-03-16-01-23.log
5 log_2016-03-16-01-24.log
6 log_2016-03-16-01-25.log
7 log_2016-03-16-01-26.log
```


■ Transformations on DStreams:

map(func)	Return a new DStream by passing each element of the source DStream through a function <i>func</i> .
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(func)	Return a new DStream by selecting only the records of the source DStream on which <i>func</i> returns true.
repartition(numPartitions)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(otherStream)	Return a new DStream that contains the union of the elements in the source DStream and <i>otherDStream</i> .
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(func)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function <i>func</i> (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
countByKey()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(func, [numTasks])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property <code>spark.default.parallelism</code>) to do the grouping. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.
join(otherStream, [numTasks])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(otherStream, [numTasks])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
transform(func)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
updateStateByKey(func)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

■ Transformations on DStreams:

UpdateStateByKey Operation (key, value)

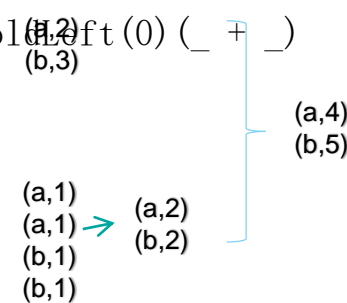
- The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do two steps.
 1. **Define the state** - The state can be an arbitrary data type.
 2. **Define the state update function** - Specify with a function how to update the state using the previous state and the new values from an input stream.

■ 举例:

```
val updateFunc = (values: Seq[Int], state: Option[Int]) => {
    val currentCount = values.foldLeft(0) (_ + _)
    val previousCount = state.getOrElse(0)
    Some(currentCount + previousCount)
}

val lines = ssc.socketTextStream(ip, port)
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val stateDstream = wordDstream.updateStateByKey[Int](updateFunc)
stateDstream.print()
```



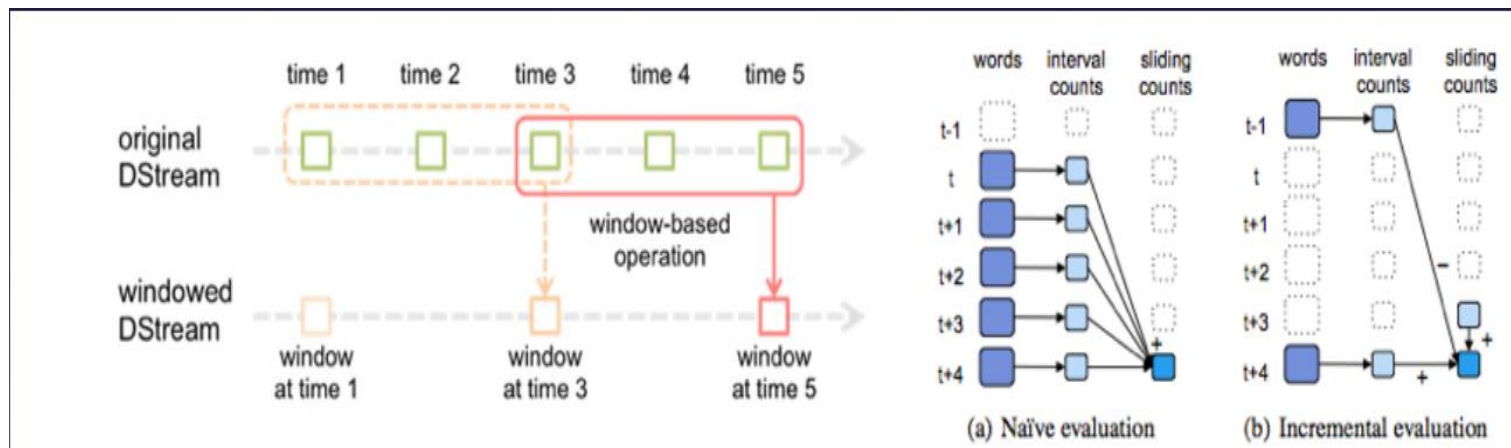
■ Transformations on DStreams:

Window Operations

- allow you to apply transformations over a sliding window of data.

window length - The duration of the window.

sliding interval - The interval at which the window operation is performed.



■ Transformations on DStreams:

Window Operations

■ 举例:

```
val wordCounts = words.map(x => (x , 1)).  
  reduceByKeyAndWindow((x : Int, y : Int) => x + y, Seconds(30), Seconds(12))
```

```
val wordCounts = words.map(x => (x , 1)).  
  reduceByKeyAndWindow(_+_, _-_, Seconds(30), Seconds(12))
```

```
words = lines.map(lambda line: getEvent(line)).groupByKey()  
windowDStream = words.window(86400, 4)
```

■ Transformations on DStreams:

Join Operations

Stream-stream joins

```
val stream1: DStream[String, String] = ...  
val stream2: DStream[String, String] = ...  
val joinedStream = stream1.join(stream2)
```

基于window的

```
val windowedStream1 = stream1.window(Seconds(20))  
val windowedStream2 = stream2.window(Minutes(1))  
val joinedStream = windowedStream1.join(windowedStream2)
```

■ Transformations on DStreams:

Transform Operations

The transform operation (along with its variations like transformWith) allows arbitrary RDD-to-RDD functions to be applied on a DStream. It can be used to apply any RDD operation that is not exposed in the DStream API. For example, the functionality of joining every batch in a data stream with another dataset is not directly exposed in the DStream API. However, you can easily use transform to do this

举例:

```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing spam information

val cleanedDStream = wordCounts.transform(rdd => {
    rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data
    cleaning
    ...
})
```

■ Output Operations on DStreams:

Output Operation	Meaning
<code>print()</code>	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called <code>pprint()</code> in the Python API.
<code>saveAsTextFiles(prefix, [suffix])</code>	Save this DStream's contents as text files. The file name at each batch interval is generated based on <i>prefix</i> and <i>suffix</i> : " <i>prefix-TIME_IN_MS[.suffix]</i> ".
<code>saveAsObjectFiles(prefix, [suffix])</code>	Save this DStream's contents as <code>SequenceFiles</code> of serialized Java objects. The file name at each batch interval is generated based on <i>prefix</i> and <i>suffix</i> : " <i>prefix-TIME_IN_MS[.suffix]</i> ". Python API This is not available in the Python API.
<code>saveAsHadoopFiles(prefix, [suffix])</code>	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on <i>prefix</i> and <i>suffix</i> : " <i>prefix-TIME_IN_MS[.suffix]</i> ". Python API This is not available in the Python API.
<code>foreachRDD(func)</code>	The most generic output operator that applies a function, <i>func</i> , to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function <i>func</i> is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

■Output Operations on DStreams:

Design Patterns for using

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection() // executed at the
  driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

这种方式不对，这种方式可能会要求，这个connection对象，能够被序列化，并且从driver端发送到worker端，但是大多数connection对象都是不能被序列化的，所以很可能会报无法序列化等问题

```
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```

这种方式，connection对象是在worker端被创建的，所以不存在发送，但是不能被序列化的问题。但是，这样要求每个rdd的元素，都会创建新的connection对象，导致集群的负荷会比较大

■Output Operations on DStreams:

Design Patterns for using foreachRDD

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val connection = createNewConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    connection.close()
  }
}
```

这种方式，是每个rdd的partition会创建新的connection对象，相比较前一种，更优化了些

最好的方式，就是设置一个连接池，创建connection对象的个数，就可以控制了

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // ConnectionPool is a static, lazily initialized pool of connections
    val connection = ConnectionPool.getConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    ConnectionPool.returnConnection(connection) // return to the pool for
    future reuse
  }
}
```

■Caching / Persistence

- 和RDD相似，DStreams也允许开发者持久化流数据到内存中。在DStream上使用 `persist()` 方法自动地持久化DStream中的RDD到内存中。
- 如果DStream中的数据需要计算多次，这是非常有用的。
- 像 `reduceByWindow` 和 `reduceByKeyAndWindow` 这种窗口操作、`updateStateByKey` 这种基于状态的操作，持久化是默认的，不需要开发者调用 `persist()` 方法。
- 例如通过网络（如kafka，flume等）获取的输入数据流，默认的持久化策略是复制数据到两个不同的节点以容错。（热备）

■Caching / Persistence

通过传递一个StorageLevel对象给persist()方法设置这些存储级别。

Storage Level	Meaning
MEMORY_ONLY	将RDD作为非序列化的Java对象存储在jvm中。如果RDD不适合存在内存中，一些分区将不会被缓存，从而在每次需要这些分区时都需重新计算它们。这是系统默认的存储级别。
MEMORY_AND_DISK	将RDD作为非序列化的Java对象存储在jvm中。如果RDD不适合存在内存中，将这些不适合存在内存中的分区存储在磁盘中，每次需要时读出它们。
MEMORY_ONLY_SER	将RDD作为序列化的Java对象存储（每个分区一个byte数组）。这种方式比非序列化方式更节省空间，特别是用到快速的序列化工具时，但是会更耗费cpu资源—密集的读操作。
MEMORY_AND_DISK_SER	和MEMORY_ONLY_SER类似，但不是在每次需要时重复计算这些不适合存储到内存中的分区，而是将这些分区存储到磁盘中。
DISK_ONLY	仅仅将RDD分区存储到磁盘中
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	和上面的存储级别类似，但是复制每个分区到集群的两个节点上面
OFF_HEAP (experimental)	以序列化的格式存储RDD到Tachyon中。相对于MEMORY_ONLY_SER，OFF_HEAP减少了垃圾回收的花费，允许更小的执行者共享内存池。这使其在拥有大量内存的环境下或者多并发应用程序的环境中具有更强的吸引力。

■Checkpointing

- Metadata checkpointing: 保存流计算的定义信息到容错存储系统如HDFS中。

元数据包括

- Configuration : 创建Spark Streaming应用程序的配置信息
 - DStream operations : 定义Streaming应用程序的操作集合
 - Incomplete batches: 操作存在队列中的未完成的批
- Data checkpointing : 保存生成的RDD到可靠的存储系统中, 这在有状态 transformation (如结合跨多个批次的数据) 中是必须的。在这样一个 transformation 中, 生成的RDD依赖于之前 批的RDD, 随着时间的推移, 这个依赖链的长度会持续增长。在恢复的过程中, 为了避免这种无限增长。有状态的 transformation 的中间RDD将会定时地存储到可靠存储系统中, 以截断这个依赖链。

元数据checkpoint主要是为了从driver故障中恢复数据。

如果transformation操作被用到了, 数据checkpoint即使在简单的操作中都是必须的。

■Checkpointing

何时checkpoint

应用程序在下面两种情况下必须开启checkpoint

- 使用有状态的transformation。如果在应用程序中用到了updateStateByKey或者reduceByKeyAndWindow，checkpoint目录必需提供用以定期checkpoint RDD。
- 从运行应用程序的driver的故障中恢复过来。使用元数据checkpoint恢复处理信息。

注意，没有前述的有状态的transformation的简单流应用程序在运行时可以不开启checkpoint。在这种情况下，从driver故障的恢复将是部分恢复（接收到了但是还没有处理的数据将会丢失）。这通常是可以接受的，许多运行的Spark Streaming应用程序都是这种方式。

怎样配置Checkpointing

以通过streamingContext.checkpoint(checkpointDirectory) 方法来做