千锋教育
www.mobiletrain.org

千 锋 强 力 推 出

逆战班

停课不停学　品质不打折

鼓励全国学子在疫情中坚持学习

好程序员
从平凡到卓越，为梦想而拼搏
GoodProgrammer.org

好程序员大数据学院出品

Spark 阶段知识串讲之SparkStreaming

DESIGN BY Goodprogrammer

第四章

Spark Streaming 回顾

第四部分 Spark Streaming 回顾

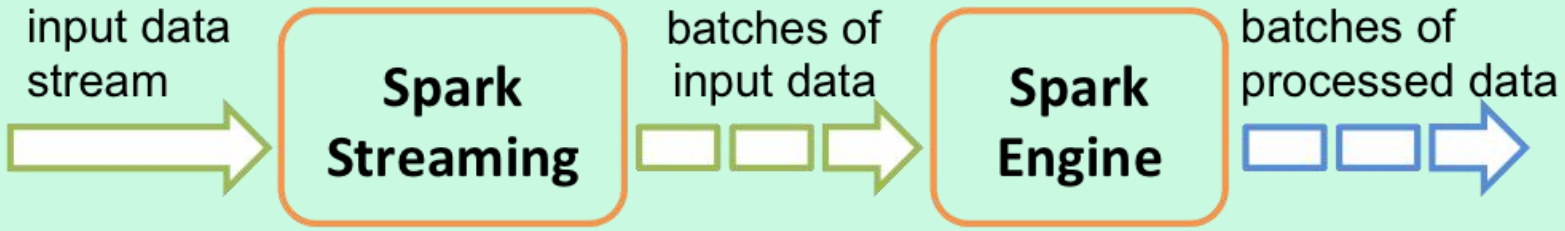
概述 →

1、Spark Streaming 基本概念
2、Spark Streaming 数据源
3、无状态转换 与 有状态转换
4、Spark Streaming 的输出操作
5、Spark Streaming 与 Kafka整合
6、offset的管理
7、Spark Streaming应用Driver进程的HA
8、Spark Streaming的面试题

1、Spark Streaming 基本概念

Spark Streaming 是基于spark的流式批处理引擎；

其基本原理是：将实时输入数据流以时间片为单位进行拆分，然后经Spark引擎以类似批处理的方式处理每个时间片数据；



Spark Streaming最主要的抽象是 Dstream(Discretized Stream，离散化数据流)，表示连续不断的数据流；

Spark Streaming提供了一个高层抽象，称为discretized stream或DStream，它表示连续的数据流。 DStream可以通过Socket、Kafka、Flume等来源的输入数据流创建，也可以通过在其他 DStream 上应用高级操作来创建。在内部，DStream表示为一系列RDD，对DStream的操作都最终转变为对相应的RDD的操作。

第四部分 Spark Streaming 回顾

1、Spark Streaming 基本概念

离散流（discretized stream）或DStream：是Spark Streaming对内部持续的实时数据流的抽象描述，即处理的一个实时数据流，在Spark Streaming中对应于一个 DStream 实例；

批数据（batch data）：这是化整为零的第一步，将实时流数据以时间片为单位进行分批，将流处理转化为时间片数据的批处理。随着持续时间的推移，这些处理结果就形成了对应的结果数据流了；

时间片或批处理时间间隔（ batch interval）：这是人为地对流数据进行定量的标准，以时间片作为我们拆分流数据的依据。一个时间片的数据对应一个RDD实例；

窗口长度（window length）：一个窗口覆盖的流数据的时间长度。必须是批处理时间间隔的倍数；

滑动时间间隔：前一个窗口到后一个窗口所经过的时间长度。必须是批处理时间间隔的倍数；

Input DStream :一个input DStream是一个特殊的DStream，将Spark Streaming连接到一个外部数据源来读取数据。

2、Spark Streaming 数据源

基本数据源：文件、socket、RDD队列流（StreamingContext实例.queueStream(...)）
高级数据源：Kafka、flume

3、无状态转换 与 有状态转换

无状态转化：每次计算的时间，仅仅计算当前时间切片的内容，每个批次处理都不依赖于先前批次的数据。

DStream无状态转化操作的例子（不完整列表）

函数名称	目　　的	Scala示例	用来操作DStream[T]的用户自定义函数的函数签名
map()	对 DStream 中的每个元素应用给定函数，返回由各元素输出的元素组成的 DStream。	ds.map(x => x + 1)	f: (T) -> U
flatMap()	对 DStream 中的每个元素应用给定函数，返回由各元素输出的迭代器组成的 DStream。	ds.flatMap(x => x.split(" "))	f: T -> Iterable[U]
filter()	返回由给定 DStream 中通过筛选的元素组成的 DStream。	ds.filter(x => x != 1)	f: T -> Boolean
repartition()	改变 DStream 的分区数。	ds.repartition(10)	N/A
reduceByKey()	将每个批次中键相同的记录归约。	ds.reduceByKey((x, y) => x + y)	f: T, T -> T
groupByKey()	将每个批次中的记录根据键分组。	ds.groupByKey()	N/A

3、无状态转换 与 有状态转换

有状态转化：依赖之前的批次数据或者中间结果来计算当前批次的数据，不断的把当前的计算和历史时间切片的RDD进行累计。

计算某个单词出现的次数，需要把当前的状态与历史的状态相累加，随着时间的流逝，数据规模会越来越大，包括updateStatebyKey、mapWithState、window操作；

有状态的转换操作需要checkpoint的支持；

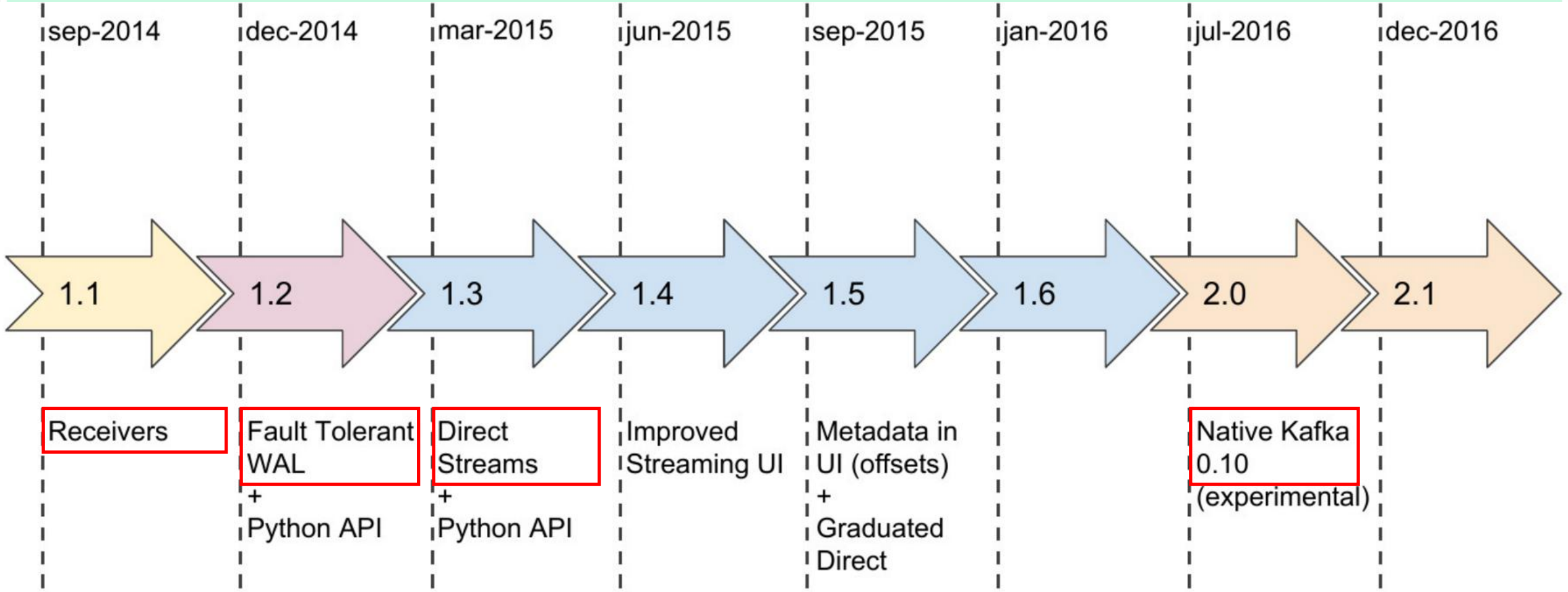
4、Spark Streming 的输出操作

Output Operations将DStream的数据推送到外部系统，如数据库或文件系统。类似于RDD的惰性求值，输出操作才会触发计算的实际执行。

Output Operation	Meaning
print()	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 **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as `SequenceFiles` of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, 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 *func* 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.

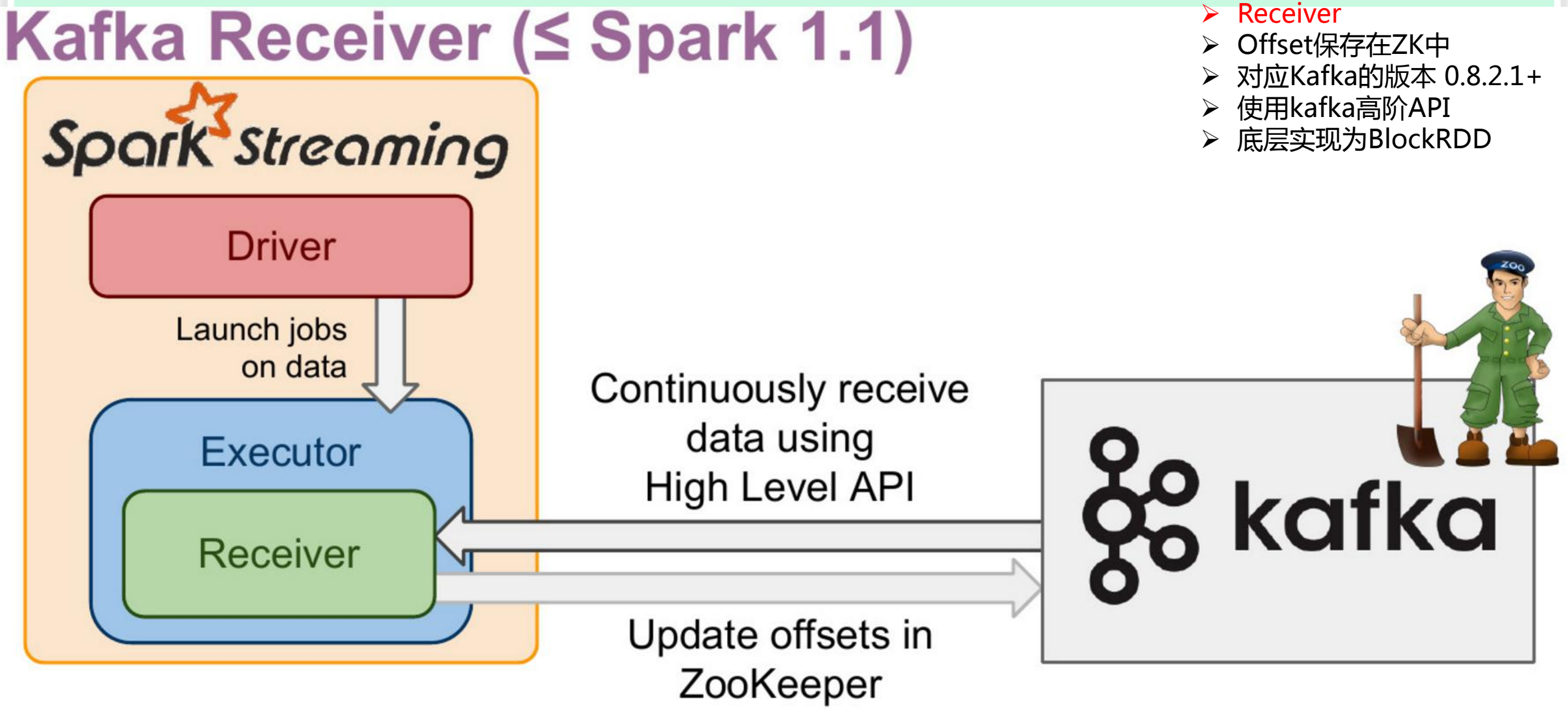
5、Spark Streaming 与 Kafka整合

对Kafka的支持分为两个版本082、010



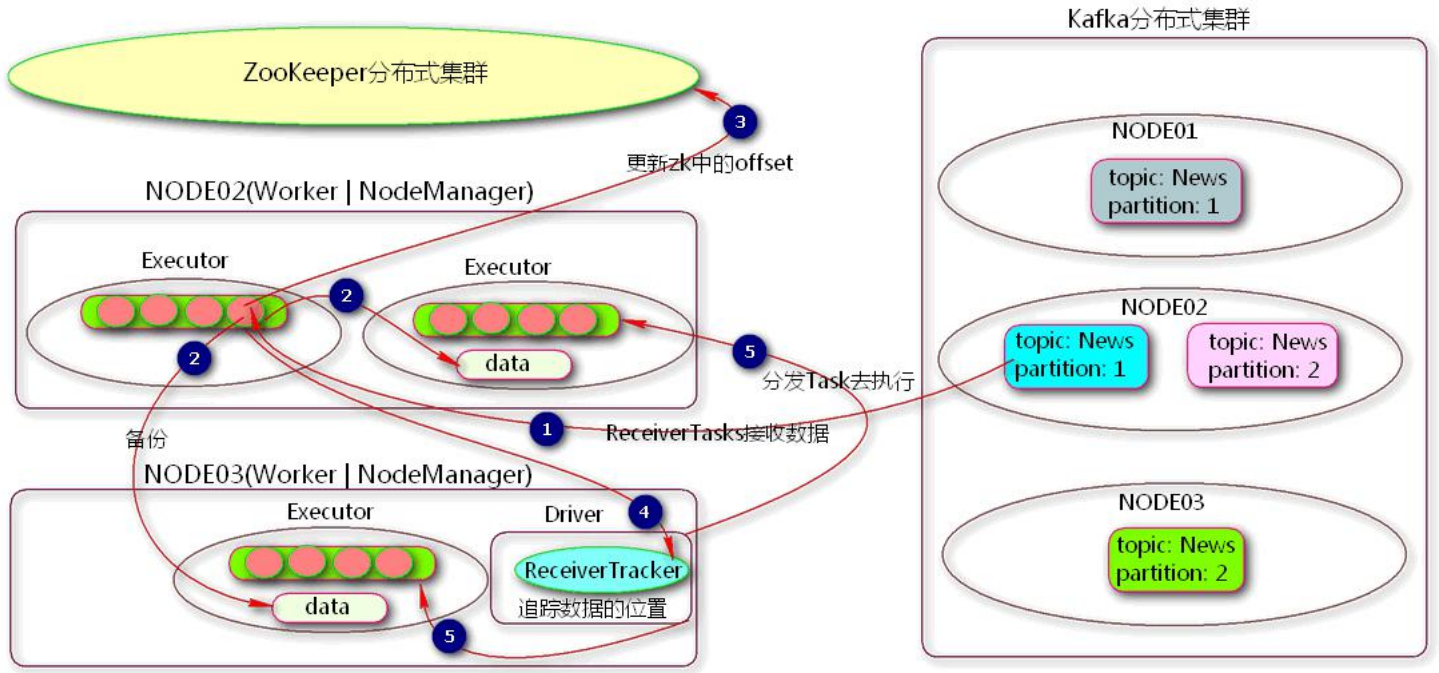
5、Spark Streaming 与 Kafka整合（续）

- Kafka 082 接口
- Receiver
- Offset保存在ZK中
- 对应Kafka的版本 0.8.2.1+
- 使用kafka高阶API
- 底层实现为BlockRDD



5、Spark Streaming 与 Kafka整合（续）

Spark Streaming + Kafka之Receiver模式



说明：

步骤2➡接收来的数据存储级别为MEMORY_AND_DISK_SER_2

步骤4➡将备份数据的位置汇报给Driver中的ReceiverTracker。Driver就知悉了data的位置。

5、Spark Streaming 与 Kafka整合（续）

→ receiver模式理解：

1）在SparkStreaming程序运行起来后，Executor中会有receiver tasks接收kafka推送过来的数据。数据会被持久化，默认级别为MEMORY_AND_DISK_SER_2,这个级别也可以修改。receiver task对接收过来的数据进行存储和备份，这个过程会有节点之间的数据传输。备份完成后去zookeeper中更新消费偏移量，然后向Driver中的receiver tracker汇报数据的位置。最后Driver根据数据本地化将task分发到不同节点上执行。

2）该模式下：

①在executor上会有receiver从kafka接收数据并存储在Spark executor中，在到了batch时间后触发job去处理接收到的数据，1个receiver占用1个core；

②为了不丢数据需要开启WAL机制，这会将receiver接收到的数据写一份备份到第三方系统上（如：HDFS）；

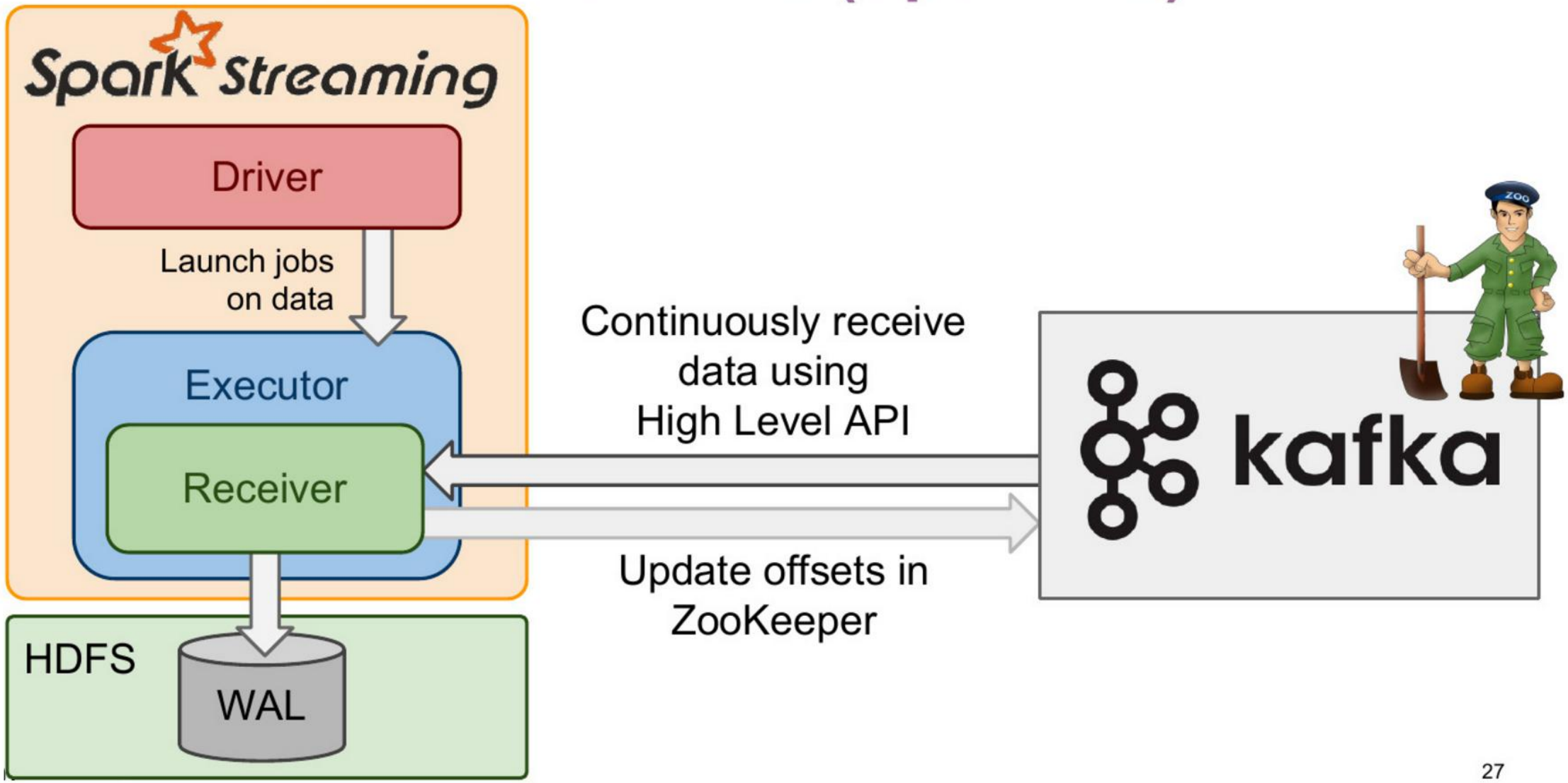
③receiver内部使用kafka High Level API去消费数据及自动更新offset。

→receiver模式中存在的问题：

当Driver进程挂掉后，Driver下的Executor都会被杀掉，当更新完zookeeper消费偏移量的时候，Driver如果挂掉了，就会存在找不到数据的问题，相当于丢失数据。

Kafka Receiver with WAL (Spark 1.2)

- Receiver with WAL



第四部分 Spark Streaming 回顾

→ 如何解决数据丢失的问题？

开启WAL(write ahead log)预写日志机制,在接受过来数据备份到其他节点的时候，同时备份到HDFS上一份（我们需要将接收来的数据的持久化级别降级到MEMORY_AND_DISK），这样就能保证数据的安全性。不过，因为写HDFS比较消耗性能，要在备份完数据之后才能进行更新zookeeper以及汇报位置等，这样会增加job的执行时间，这样对于任务的执行提高了延迟度。

→ 注意：

1）开启WAL之后，接受数据级别要降级，有效率问题

2）开启WAL要checkpoint

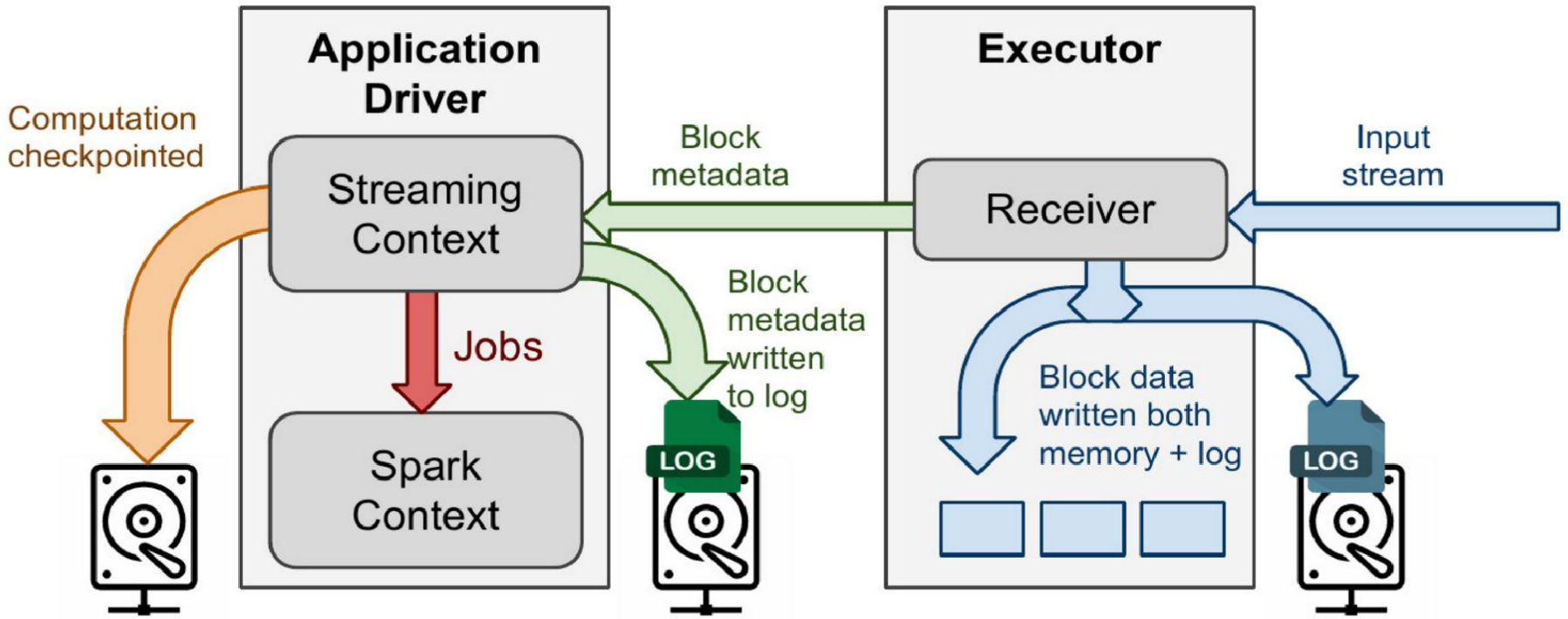
3）开启WAL(write ahead log),往HDFS中备份一份数据

→ receiver的并行度设置

receiver的并行度是由spark.streaming.blockInterval来决定的，默认为200ms,假设batchInterval为5s,那么每隔blockInterval就会产生一个block,这里就对应每批次产生RDD的partition,这样5秒产生的这个Dstream中的这个RDD的partition为25个，并行度就是25。如果想提高并行度可以减少blockInterval的数值，但是最好不要低于50ms。

第四部分 Spark Streaming 回顾

Kafka Receiver with WAL (Spark 1.2)



步骤→

1)InputDStream: 从流数据源接收的输入数据。

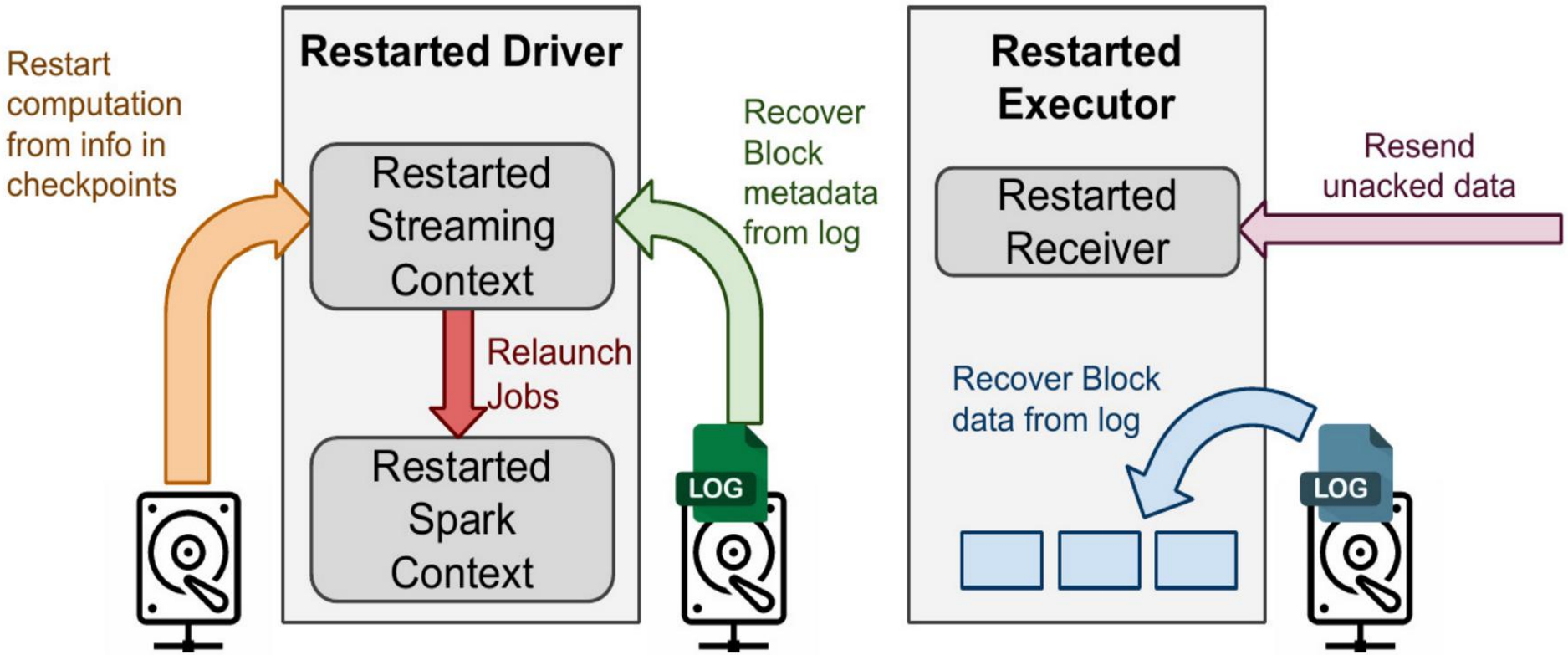
2)Receiver：负责接收数据流，并将数据写到本地。

3)StreamingContext：代表SparkStreaming，负责Streaming层面的任务调度，生成jobs发送到Spark engine处理。

4)Spark Context: 代表Spark Core，负责批处理层面的任务调度，真正执行job的Spark engine。

- 由于流计算系统是长期运行、且不断有数据流入，因此其Spark守护进程（Driver）的可靠性至关重要，它决定了Streaming程序能否一直正确地运行下去。
- Driver实现HA的解决方案就是将元数据持久化，以便重启后的状态恢复。如上图所示，Driver持久化的元数据包括：
  → ①Block元数据（上图中的绿色箭头）：Receiver从网络上接收到的数据，组装成Block后产生的Block元数据；
  → ②Checkpoint数据（上图中的橙色箭头）：包括配置项、DStream操作、未完成的Batch状态、和生成的RDD数据等。

Kafka Receiver with WAL (Spark 1.2)



→ 恢复计算（左图中的橙色箭头）：使用Checkpoint数据重启driver，重新构造上下文并重启接收器。

→ 恢复元数据块（左图中的绿色箭头）：恢复Block元数据。

→ 恢复未完成的作业（左图中的红色箭头）：使用恢复出来的元数据，再次产生RDD和对应的job，然后提交到Spark集群执行。

→ 通过如上的数据备份和恢复机制，Driver实现了故障后重启、依然能恢复Streaming任务而不丢失数据，因此提供了系统级的数据高可靠。

第四部分 Spark Streaming 回顾

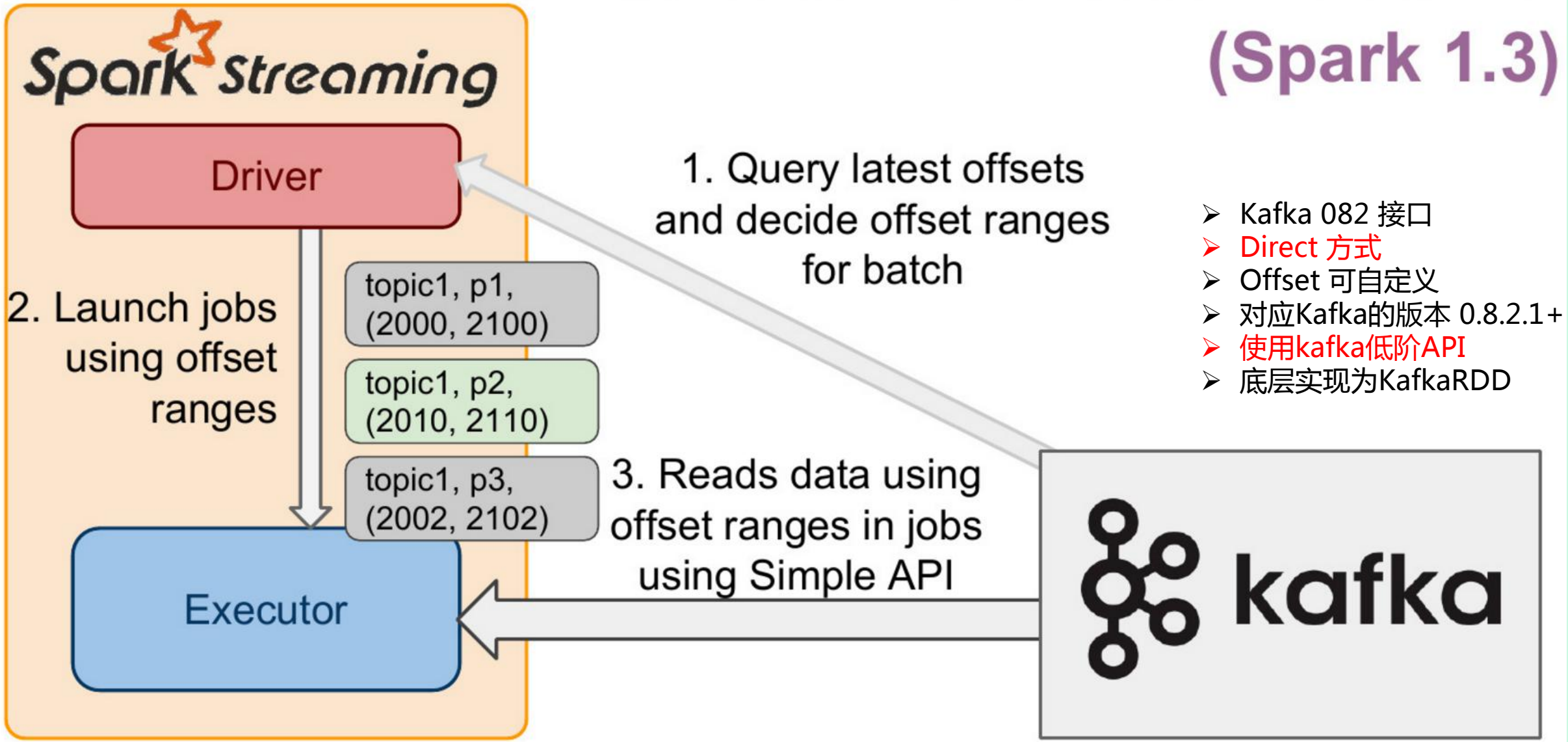
Driver HA机制 →

```
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)
    val linesDStream = ssc.socketTextStream(...)
    ssc.checkpoint(checkpoint)
    ssc
}

val context = StreamingContext.getOrCreate(checkpointy, functionToCreateContext _)
context.start()
context.awaitTermination()
```

- 进行Spark Streaming应用程序的重写后，当第一次运行程序时，如果发现checkpoint目录不存在，那么就使用定义的函数来第一次创建一个StreamingContext，并将其元数据写入checkpoint目录；当从Driver失败中恢复过来时，发现checkpoint目录已经存在了，那么会使用该目录中的元数据创建一个StreamingContext。
- 但是上面的重写应用程序的过程，只是实现Driver失败自动恢复的第一步。第二步是，必须确保Driver可以在失败时，自动被重启。
- 要能够自动从Driver失败中恢复过来，运行Spark Streaming应用程序的集群，就必须监控Driver运行的过程，并且在它失败时将它重启。对于Spark自身的standalone模式，需要进行一些配置去supervise driver，在它失败时将其重启。
- 首先，要在spark-submit中，添加--deploy-mode参数，默认其值为client，即在提交应用的机器上启动Driver；但是，要能够自动重启Driver，就必须将其值设置为cluster；此外，需要添加--supervise参数。
- 使用上述提交应用之后，就可以让driver在失败时自动被重启，并且通过checkpoint目录的元数据恢复StreamingContext。

第四部分 Spark Streaming 回顾



(Spark 1.3)

- Kafka 082 接口
- Direct 方式
- Offset 可自定义
- 对应Kafka的版本 0.8.2.1+
- 使用kafka低阶API
- 底层实现为KafkaRDD

第四部分 Spark Streaming 回顾

1、Direct模式理解

1）与receiver模式类似，不同在于executor中没有receiver组件，从kafka拉去数据的方式不同。

2）SparkStreaming+kafka 的Driect模式就是将kafka看成存数据的一方，不是被动接收数据，而是主动去取数据。消费者偏移量也不是用zookeeper来管理，而是SparkStreaming内部对消费者偏移量自动来维护，默认消费偏移量是在内存中，当然如果设置了checkpoint目录，那么消费偏移量也会保存在checkpoint中。当然也可以实现用zookeeper来管理。

3）注意点：

①没有receiver，无需额外的core用于不停地接收数据，而是定期查询kafka中的每个partition的最新的offset，每个批次拉取上次处理的offset和当前查询的offset的范围的数据进行处理；

②为了不丢数据，无需将数据备份落地，而只需要手动保存offset即可；

③内部使用kafka simple Level API去消费数据, 需要手动维护offset，kafka zk上不会自动更新offset。

2、Direct模式并行度设置

Direct模式的并行度是由读取的kafka中topic的partition数决定的。

第四部分 Spark Streaming 回顾

Note: Kafka 0.8 support is deprecated as of Spark 2.3.0.

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
API Maturity	Deprecated	Stable
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic Topic Subscription	No	Yes

第四部分 Spark Streaming 回顾

spark-streaming-kafka-0-8

- Receiver方式
  - ✓ Kafka高阶API
  - ✓ Offset系统管理，保存在ZooKeeper中
  - ✓ 为了减少数据的丢失提供了WAL方式
- Direct方式
  - Kafka低阶API
  - Offset管理自定义管理方式

Receiver 方式 与 Direct方式的比较→

1、前者在executor中有Receiver接受数据，并且1个Receiver占用一个core；而后者无Receiver，减少不必要的cpu占用；

2、为了保证不丢失数据，前者需要开启WAL机制；而后者不需要，只需要在程序中成功消费完数据后再更新偏移量即可，进一步减少磁盘读写；

3、前者使用zookeeper来维护consumer的偏移量；后者通过手动维护offset，实现精确的一次消费；

4、前者InputDStream的分区是 num_receiver *batchInterval/blockInteral，后者的分区数是kafka topic partition的数量，也就是DStream中生成的RDD与kafka分区一一对应，便于把控并行度。（Receiver模式下num_receiver的设置不合理会影响性能或造成资源浪费；如果设置太小，并行度不够，整个链路上接收数据将是瓶颈；如果设置太多，则会浪费资源）

6、Offset 的管理（spark-streaming-kafka-0-10）

系统管理的offset。对于一些streaming应用程序，如实时活动监控，只需要当前最新的数据，这种情况不需要管理offset；

外部存储保存offset。

- Checkpoint
- HDFS
- ZooKeeper
- HBase
- Kafka
- RDBMS
- Redis

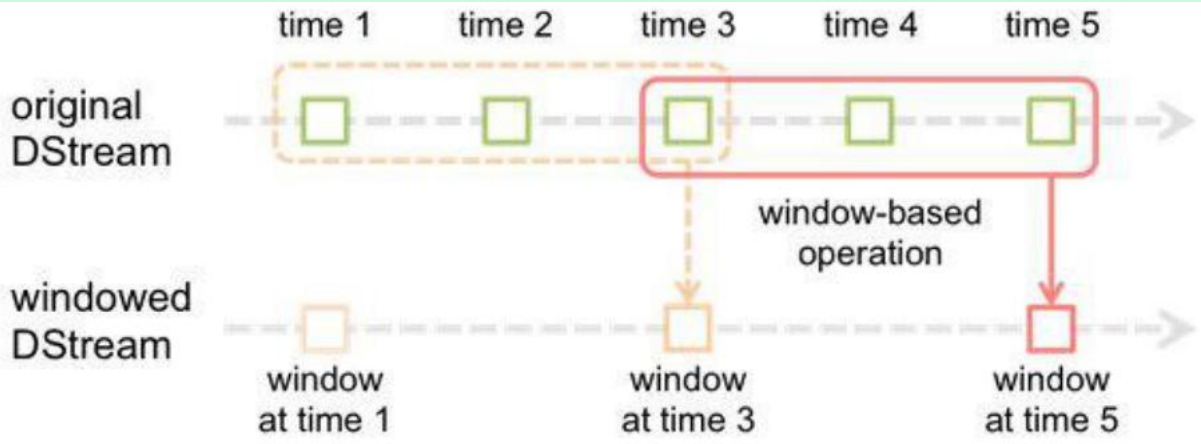
1、阐述DStream的原理

DStream是Spark Streaming的基础抽象，代表持续的数据流，它由一系列连续的RDD组成，一个批次间隔接收的数据只会存放在一个block中，因此每个批次间隔都只会产生一个RDD。

DStream与RDD同样是不可变的，每个算子都会创建一个新的DStream，因此一个批次可能会有多个DStream。

对同一个DStream连续window没有意义，因为foreach只会按照最后一个window生成的DStream来对待RDD。

2、 窗口



- 窗口的模型→
- 假设批次间隔interval=1，窗口长度length=3，滑动间隔window_interval=2,每个批次数据为A,B,C,D,E,F,G。则统计的窗口为(A,B,C),(C,D,E),(E,F,G)
- 1. 一开始先按interval的速度一路往前，直到走过length为止停下，执行第一个foreachRDD，因此第一个RDD是基于批次间隔的
- 2. 接着每经过一个window_interval，就将从终点往前的length长度纳入本次的统计范围，但是实质上DStream只走了window_interval的距离，因此从第二个RDD开始都是基于滑动间隔的。
- 每个DStream实质上都是个窗口，只不过初始的DStream是基本窗口，它的窗口大小=滑动间隔=批次间隔。

2、 窗口（续）

```
/**
  * 推荐使用reduceByKeyAndWindow+foreachRDD进行窗口聚合操作：
  *
  * @param  mergeFunc 对窗口期间的数据进行聚合的方法，只传入该函数可以直接实现聚合
  * @param  invFunc   利用上个窗口聚合的数据进行统计的高效算法，步骤如下：
  *             1.先执行mergeFunc方法统计本次滑动间隔产生的数据nowValue
  *             2.把新数据与上批数据累加：newValue = nowValue+oldValue
  *             3.调用用户传入的invFunc方法，只保留两个窗口间重复的数据：invFunc(newValue - 上个批次的滑动数据)
 */
 reduceByKeyAndWindow（mergeFunc,invFunc,windowLength,slideInterval）
```

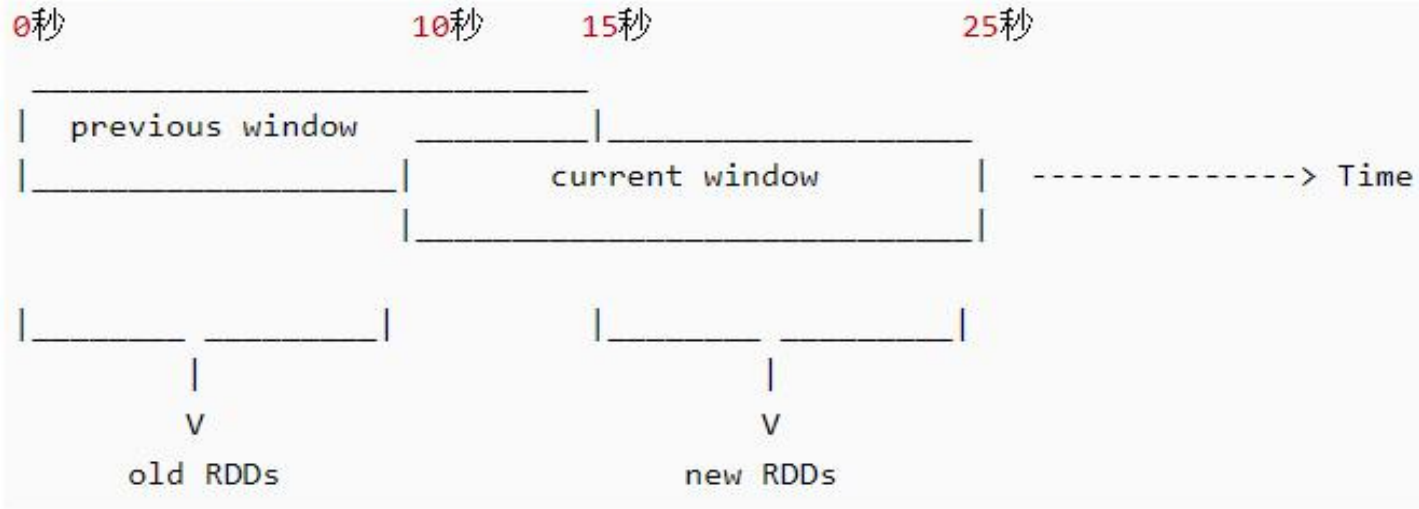
注意：

当滑动间隔=窗口长度时，传入invFunc是没有意义的，但如果两者不相等，传入invFunc对性能的提升很大。

2、 窗口（续）

最终结果 = 重复区间（previous window的值 - oldRDD的值） 也就是中间重复部分 + newRDD的值

→ 结果就是10秒到25秒这个时间区间的值



reduceByKeyAndWindow(_+_, _-_, Duration, Duration)

步骤：

1. 存储上一个window的reduce值
2.计算出上一个window的begin 时间到 重复段的开始时间的reduce 值 → oldRDD
3.重复时间段的值结束时间到当前window的结束时间的值 → newRDD
4.重复时间段的值等于上一个window的值减去oldRDD

注意：函数reduceByKeyAndWindow（_+_,Seconds(3), Seconds(2)）的重载函数reduceByKeyAndWindow（_+_,_-_,Seconds(3s),seconds(2)）

设计理念→当 滑动窗口的时间Seconds(2) < Seconds(3)（窗口大小）时，两个统计的部分会有重复，那么我们就可以不用重新获取或者计算，而是通过获取旧信息来更新新的信息，这样即节省了空间又节省了内容，并且效率也大幅提升。

3、 SparkStreaming如何对接Kafka

方式1：receiver

sparkStream启动一个单独的线程receiver定时使用kafka高阶API向kafka拉取数据，并自动地更新zk的offsets。

→ 优点：用户专注于业务，不需要关心偏移量的维护，代码简洁。

→ 缺点：定时拉取数据可能造成sparkStream处理速度跟不上，导致数据丢失。

启动wal预写日志后，receiver会额外将数据写一份到本地，数据丢失的情况可以自动到日志中恢复，但是这种方式会重复写数据造成性能大幅浪费。此外，receiver与业务不在同一线程，但两者却又相互依赖，这导致我们在对业务进行高并发高吞吐的优化时不得不受制于receiver。

方式2：direct

sparkStream在业务代码中使用kafka低阶API直接连接kafka拉取数据进行消费。

→ 优点：

简化并行：kafka分区与RDD分区一致，可以一对一并行消费;

高效：数据的拉取与消费是顺序关系，不存在数据丢失问题，避免wal预写日志

稳定：处理完才拉取下一批数据，不会造成任务积压导致程序崩溃

强一致语义：可以通过手动维护偏移量的方式自定义实现一致性。

→ 缺点：需要采用checkpoint或第三方平台维护偏移量，开发成本较高;实现监视需要额外人工开发。

3、 SparkStreaming如何解决数据积压

→ 起因

sparkStream作为一个微流处理框架，每批次处理数据的时间应尽可能地接近批次间隔时间，才能保证流处理的高效和稳定。

批处理时间<批间隔时间：流量太小，集群闲置，浪费资源

批处理时间>批间隔时间：流量太大，集群繁忙，数据积压导致系统崩溃

→ 流量控制

通过设置spark.streaming.kafka.maxRatePerPartition可以静态调整每次拉取的最大流量，但是需要重启集群。

→ 反压机制

不需要重启集群就能根据当前系统的处理速度智能地调节流量阈值的方案

设置spark.streaming.backpressure.enabled为true开启反压机制后，sparkStreaming会根据上批次和本批次的处理速率，自动估算出下批次的流量阀值，我们可以通过改变几个增益比例来调控它的自动估算模型。

它的底层采用的是Guava的令牌桶算法实现的限流：程序到桶里取令牌，如果取到令牌就缓存数据，取不到就阻塞等待。通过改变放令牌的速度即可实现流量控制。

→ 其它方案

1. 如果增加kafka的分区数，spark也会增加相应数目的消费者去拉取，可以提升拉取效率;

2. 如果降低批次间隔时间，每次拉取的数据量会减少，可以提升处理数据的速度，差距的间隔时间可以通过窗口来弥补。

4、 SparkStreaming如何实现批次累加

→ updataStateBykey

updataStateBykey是特殊的reduceByKey，相当于oldValue+reduceByKey(newValue1,newValue2)，通过传入一个updateFunc来实现批次间数据累加的操作。

实现它必须设置checkPoint路径，updataStateBykey会自动将每次计算的结果持久化到磁盘，批次间的数据则是缓存在内存中。

缺点：大量占用内存，大量产生小文件

→ mapwithState

mapwithState是spark1.6新增的累加操作，目前还在测试中，它的原理网上查不到，只知道是updataStateBykey的升级版，效率提升10倍。

缺点：资料不全，社区很小

不建议使用状态流累加操作，建议用窗口+第三方存储(redis)来达到同样的效果。

→Spark Streaming中的updateStateByKey和mapWithState的区别和使用

①**UpdateStateByKey**：统计全局的key的状态，但是就算没有数据输入，他也会在每一个批次的时候返回之前的key的状态。这样的缺点就是，如果数据量太大的话，而且我们需要checkpoint数据，这样会占用较大的存储。

如果要使用updateStateByKey,就需要设置一个checkpoint目录（updateStateByKey自己是无法保存key的状态的），开启checkpoint机制。因为key的state是在内存维护的，如果宕机，则重启之后之前维护的状态就没有了，所以要长期保存它的话需要启用checkpoint，以便恢复数据。

4、 SparkStreaming如何实现批次累加（续）

→Spark Streaming中的updateStateByKey和mapWithState的区别和使用（续）

①**MapWithState**：也是用于全局统计key的状态，但是它如果没有数据输入，便不会返回之前的key的状态，有一点增量的感觉。

这样做的好处是，我们可以只关心那些已经发生变化的key，对于没有数据输入，则不会返回那些没有变化的key的数据。这样即使数据量很大，checkpoint也不会像updateStateByKey那样，占用太多的存储。

5、SparkStreaming一个批次有多久？一个批次有多少条数据？

关于批次间隔需要结合业务来确定的，如果实时性要求高，批次间隔需要调小。

每个批次的数据量是和每天产生的数据量有直接关系，在计算的时候需要考虑峰值的情况。需要注意的是，批次间隔越长，每个批次计算的数据量会越多。

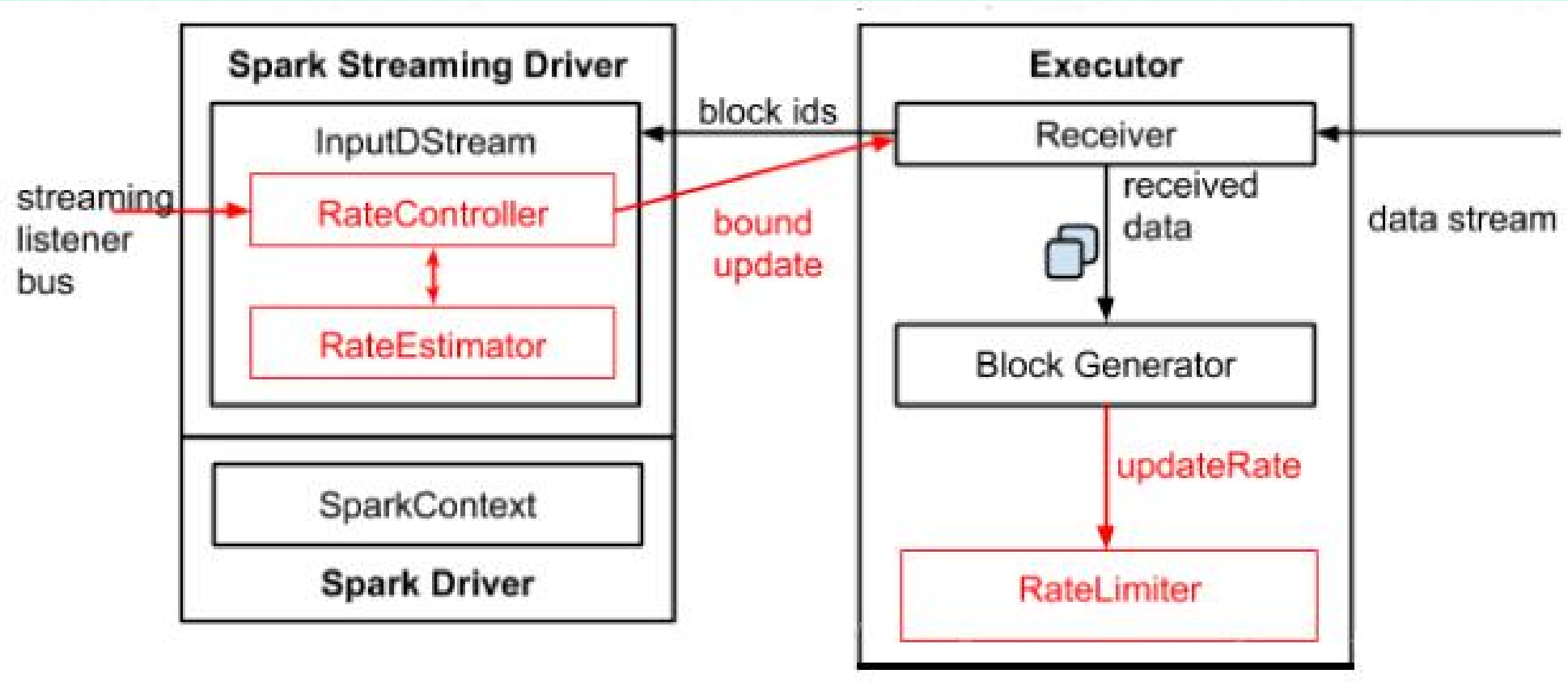
6、SparkStreaming消费速度赶不上生产速度怎么办？

在默认情况下，Spark Streaming 通过receiver或者Direct方式以生产者生产数据的速率接收数据。当 batch processing time > batch interval 的时候，也就是每个批次数据处理的时间要比 Spark Streaming 批处理间隔时间长。越来越多的数据被接收，但是数据的处理速度没有跟上，导致系统开始出现数据堆积，可能进一步导致 Executor 端出现 OOM 问题而出现失败的情况。

Spark Streaming 1.5 之后的体系结构：

6、SparkStreaming消费速度赶不上生产速度怎么办？（续）

Spark Streaming 1.5 之后的体系结构：（续）



6、SparkStreaming消费速度赶不上生产速度怎么办？（续）

● 为了实现自动调节数据的传输速率，在原有的架构上新增了一个名为 RateController 的组件，这个组件继承自 StreamingListener，其监听所有作业的 onBatchCompleted 事件，并且基于 processingDelay、schedulingDelay 、当前 Batch 处理的记录条数以及处理完成事件来估算出一个速率；这个速率主要用于更新流每秒能够处理的最大记录的条数。速率估算器（RateEstimator）可以又多种实现，不过目前的 Spark 2.2 只实现了基于 PID 的速率估算器。

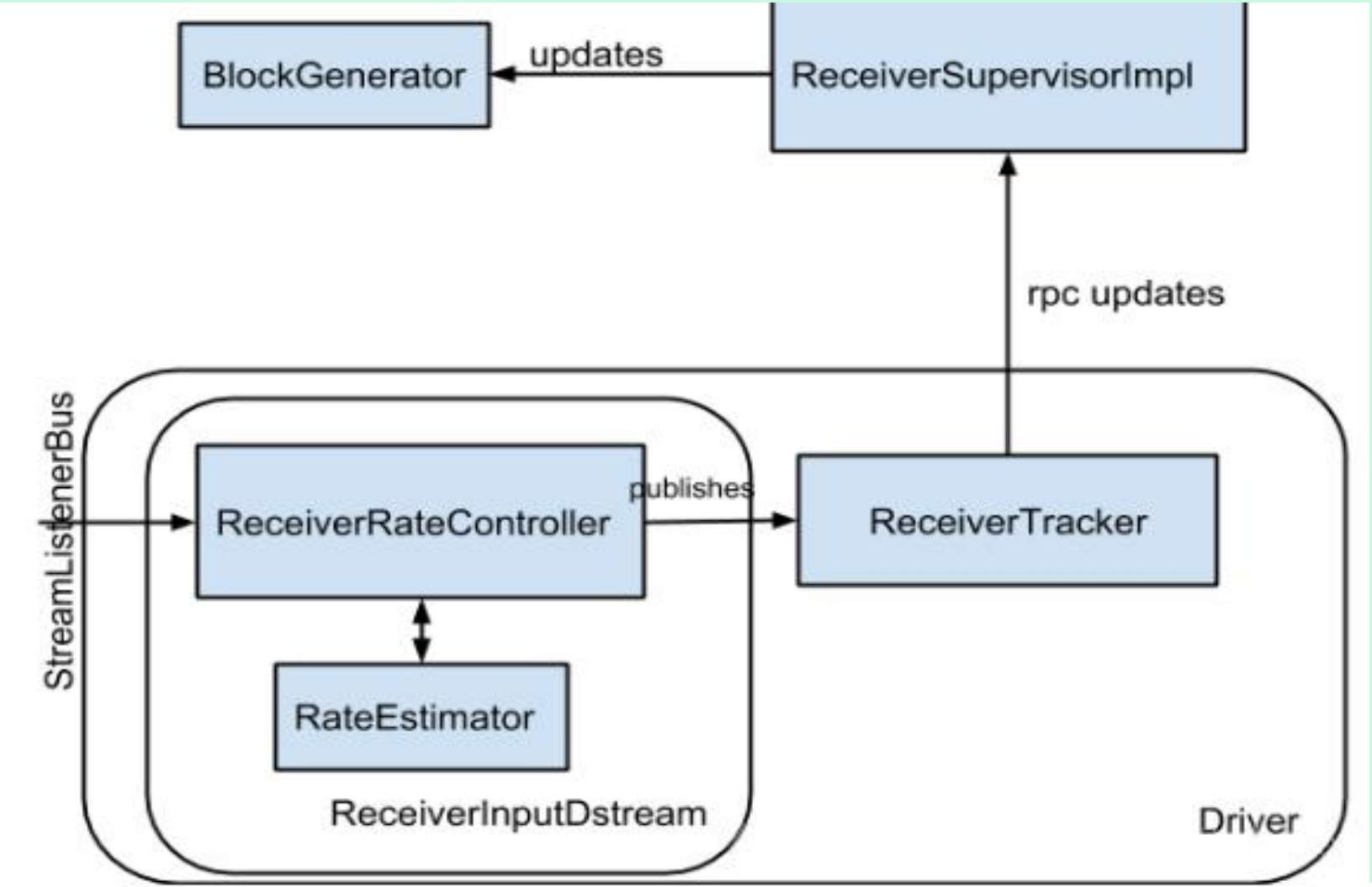
● InputDStreams 内部的 RateController 里面会存下计算好的最大速率，这个速率会在处理完 onBatchCompleted 事件之后将计算好的速率推送到 ReceiverSupervisorImpl，这样接收器就知道下一步应该接收多少数据了。

● 如果用户配置了 spark.streaming.receiver.maxRate 或 spark.streaming.kafka.maxRatePerPartition，那么最后到底接收多少数据取决于三者的最小值。也就是说每个接收器或者每个 Kafka 分区每秒处理的数据不会超过 spark.streaming.receiver.maxRate 或 spark.streaming.kafka.maxRatePerPartition 的值。

详细的过程如下图所示：

6、SparkStreaming消费速度赶不上生产速度怎么办？（续）

- 详细的过程如下图所示：（续）



6、SparkStreaming消费速度赶不上生产速度怎么办？（续）

→ 如何启用？

在 Spark 启用反压机制很简单，只需要将 spark.streaming.backpressure.enabled 设置为 true 即可，这个参数的默认值为 false。反压机制还涉及以下几个参数，包括文档中没有列出来的：

- spark.streaming.backpressure.initialRate： 启用反压机制时每个接收器接收第一批数据的初始最大速率。默认值没有设置。
- spark.streaming.backpressure.rateEstimator：速率估算器类，默认值为 pid ，目前 Spark 只支持这个，大家可以根据自己的需要实现。
- spark.streaming.backpressure.pid.proportional：用于响应错误的权重（最后批次和当前批次之间的更改）。默认值为1，只能设置成非负值。weight for response to “error” (change between last batch and this batch)
- spark.streaming.backpressure.pid.integral：错误积累的响应权重，具有抑制作用（有效阻尼）。默认值为 0.2 ，只能设置成非负值。weight for the response to the accumulation of error. This has a dampening effect.
- spark.streaming.backpressure.pid.derived：对错误趋势的响应权重。 这可能会引起 batch size 的波动，可以帮助快速增加/减少容量。默认值为0，只能设置成非负值。weight for the response to the trend in error. This can cause arbitrary/noise-induced fluctuations in batch size, but can also help react quickly to increased/reduced capacity.
- spark.streaming.backpressure.pid.minRate：可以估算的最低费率是多少。默认值为 100，只能设置成非负值。

以上为Spark的反压机制，再结合Spark资源的动态调整（在下面的题中有详细解释），就是该问题的完整解决方案。

7、SparkStreaming的批次间隔，处理完的的数据存在哪里？

批次间隔为SparkStreaming处理实时需求的时间间隔，需要根据业务需求来确定批次间隔。

实时需求的处理结果一般是保存在能快速读取的数据库中来提高效率，比如Redis、MongoDB、HBase。

8、Spark Streaming的窗口大小，每个窗口处理的数据量大小是多少？

该问题一定要根据业务需求来确定，比如要实现的需求为：统计每分钟的前一个小时的在线人数。

上面需求的窗口大小（窗口长度）为1小时，然后再统计每个窗口需要处理的数据量。

窗口处理的数据量 = 每个批次处理的平均数据量 * 窗口的批次数量

9、MySQL的数据如何被Spark Streaming消费，假如：MySQL中用户名为张三，Spark已经消费了，但是此时我的名字改为了张小三，怎么办？如何同步？

Spark Streaming是批处理，每个批次的计算方式都是从MySQL中消费到数据进行统计，得到结果后会紧接着将结果持久化到对应的数据库，此时如果MySQL的某个字段值更新了，更新的值是无法影响以前批次的Streaming的结果的，只能影响以后批次的结果。除非是将之前的结果覆盖操作。