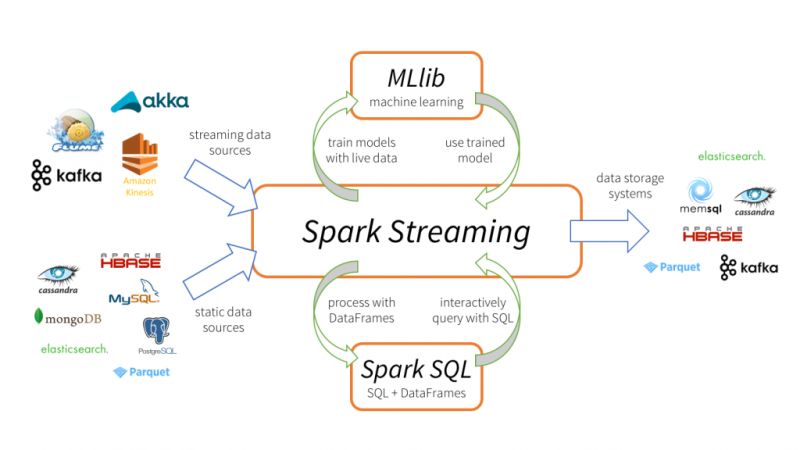
Spark Streaming

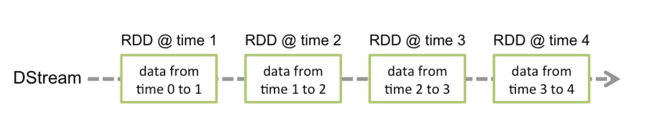
Spark Streaming是Spark 核心API的一个扩展，具有吞吐量高、容错能力强的实时流数据处理系统，支持包括Kafka、Flume、HDFS/S3、Twitter、ZeroMQ以及TCP Sockets等数据源。获取数据后可以使用Map、Reduce、Join和Window等高级函数进行复杂算法的处理，处理结果存储到文件系统、数据库或展示到仪表盘等，其中Spark Streaming数据处理流程如下图所示：



另外Spark Streaming能和Spark其他组件，如Mlib以及Graphx等融合，对实时数据进行更加复杂的处理。Spark Streaming在内部的处理机制是：先接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，这些批数据在Spark内核对应一个RDD实例，因此流数据Dstream可以看做是一组RDDs，然后调用Spark核心的作业处理这些批数据，最终得到处理后的一批批结果数据。在流数据分成一批一批后，通过一个FIFO队列，然后Spark核心作业从该队列中依次取出一个个批数据，把批数据封装成RDD，然后进行处理，典型的生产者消费者模型。

# 1.术语

1) Dstream（Discretized Stream），Spark Streaming的基础抽象，代表持续性的数据流，这些数据流可以通过外部数据源获取，也可以通过现有的Dstream转换操作来获得。在内部实现上Dstream由一组时间序列上连续的RDD表示：



在Dstream中定义了名为generatedRDDs离散数据流，以时间为键、RDD为值的哈希列表。在流数据的接收过程中，源源不断把接收到的数据放入该列表中，而对于不需要的旧RDD从该列表中删除：

*private[streaming] var generatedRDDs = new HashMap[Time, RDD[T]]()*

2) DstreamGraph，在Spark Core中，作业是由一系列具有依赖关系的RDD及作用于这些RDD上的函数所组成的操作键，在遇到Action操作时触发运行，向DAGScheduler提交并运行作业。Spark Streaming中作业的生成与Spark Core类似，对Dstream进行的各种操作让他们之间构建起依赖关系。当遇到Dstream使用输出操作时，这些依赖关系以及他们之间的操作会被记录到名为DstreamGraph的对象中表示一个作业。这些作业注册到DstreamGraph中不会立即执行，而是等到Spark Streaming启动后，到达批处理时间时，才根据DstreamGraph生成作业处理该批处理时间内接收到的数据。

在Spark Streaming中如果应用程序有多个输出操作，在批处理中会生成多个作业，这些作业的调度阶段有可能依赖，需要根据实际情况进行判断。

3）批处理间隔（Batch Duration），在Spark Streaming中，数据采集是逐条进行的，而数据处理是按批进行的，因此在Spark Streaming中会先设置好批处理间隔（batch duration）。当超过批处理间隔的时间就会把采集到的数据汇总起来成一批数据交给系统去处理。

4）窗口间隔（Window Duration）和滑动间隔（Slide Duration），对于窗口操作，批处理间隔、窗口间隔和滑动间隔是非常重要的时间概念。对于窗口操作而言，在其窗口内部会有N个批处理数据，批处理数据的个数由窗口间隔（Window Duration）决定，其为窗口持续的时间，在窗口操作中只有窗口间隔满足才会触发批处理。

除了窗口间隔，另一个重要的参数是滑动间隔，指得是多次时间窗口滑动一次行程新的窗口，默认情况下和批处理间隔相同，窗口间隔一般比滑动间隔和批处理间隔大，而且窗口间隔和滑动间隔是批处理间隔的整数倍。

# 2.Spark Streaming运行架构

Spark Streaming相对于其他流处理系统的最大优势在于流处理引擎和数据处理在同一个软件栈中，其中Spark Streaming的功能主要包括流处理引擎的流数据接收与存储以及批处理作业的生成与管理，而Spark核心负责处理Spark Streaming发送过来的作业。

Spark Streaming 分为Driver端和Client端，运行在Driver端为StreamingContext实例。该实例包括DstreamGraph和JobScheduler(包括ReceiveTracker和JobGenerator)等，而Client包括RecieverSupervisor和Reciever等。

Spark Streaming进行流数据处理大致可以分为：启动流处理引擎、接收以及存储流数据、处理流数据和输出处理结果等4个步骤，其运行架构如图所示：



（1）初始化StreamingContext对象，在该对象启动过程中实例化DstreamGraph和JobScheduler，其中DstreamGraph用于存放Dstream以及Dstream之间的依赖关系等信息，而JobScheduler中包括ReceiverTracker和JobGenerator。其中ReceiverTracker为Driver端流数据接收器（Receiver）的管理者，JobGenerator为批处理作业生成器。在ReceiverTracker的启动过程中，根据流数据接收器分发策略通知对应的Executor中流数据接收管理器（ReceiverSupervisor）启动，再由ReceiverSupervisor启动流数据接收器。

（2）当流数据接收器Reciever启动后，持续不断的接收实时流数据，根据传过来的数据大小进行判断，如果数据量很小，则攒多条数据长一块，然后进行块存储；如果数据量大，则直接进行块存储。对于这些数据Reciever直接交给RecieverSupervisor，由其进行数据转储操作。块存储根据设置是否预写日志分为两种：一种是使用非预写日志BlockManage

dBasedBlockHandler方法直接写到Worker的内存或磁盘中，另一种是进行预写日志WriteAheadLogBasedBlockHandler方法，即在预写日志同时把数据写入Worker内存或磁盘中。数据存储完毕后，RecieverSupervisor会把数据存储的元信息上报给RecieverTracker，再把这些信息转发给RecievedBlockTracker，由它负责管理收到的数据块的元信息。

（3）在StreamingContext的JobGenerator中维护一个定时器，该定时器在批处理时间到来时会进行生成作业的操作。

* 通知ReceiverTracker将接收到的数据进行提交，在提交时采用synchronized关键字进行处理，保证每条数据被划入一个且只被划入一个批中。
* 要求DstreamGraph根据Dstream依赖关系生成作业序列Seq[Job]
* 从第一步中ReceiverTracker获取本批次数据的元数据
* 把批处理时间time、作业序列Seq[Job]和本批次数据的元数据包装成JobSet，调用JobScheduler.submitJobSet(JobSet)提交给JobScheduler，JobScheduler将把这些作业发送给Spark核心进行处理，由于该执行为异步，因此执行速度非常快
* 只要提交结束，SparkStreaming对整个系统做一个检查点Checkpoint
* 在Spark核心的作业队数据进行处理，处理完毕后输出到外部系统，如数据库或文件系统，输出的数据可以被外部系统所使用。由于实时流数据源不断的流入，Spark会周而复始的进行数据处理，相应也会持续不断的输出结果

## 2.1 消息通信

下面对StreamingContext启动流处理引擎和接收存储流数据中进行的消息通信进行分析,如下图所示：



在启动流处理引擎过程中，将进行启动所有流数据接收器Receiver和注册流数据接收器Receiver两个消息通信；在接受存储流数据中，当数据存储完成后发送添加数据块消息。而当Spark Streaming停止时，需要发送关闭所有数据流数据接收器Receiver消息。

1. 在启动流处理引擎过程中，JobScheduler在内部启动ReceiverTracker和ReceiverTrackerEn

dpoint终端点，当ReceiverTracker准备完毕后向终端点发送StartAllReceivers消息，通知其分发并启动所有流数据接收器Receiver，其代码如下：

*override def receive: PartialFunction[Any, Unit] = {*

*// 根据流数据接收器分发策略，匹配流数据接收器Receiver和Executor*

*case StartAllReceivers(receivers) =>*

*val scheduledLocations = schedulingPolicy.scheduleReceivers(receivers, getExecutors)*

*for (receiver <- receivers) {*

*val executors = scheduledLocations(receiver.streamId)*

*updateReceiverScheduledExecutors(receiver.streamId, executors)*

*//在HashMap中保留流数据接收器Receiver首选位置*

*receiverPreferredLocations(receiver.streamId) = receiver.preferredLocation*

*//在指定的Executor中启动流数据接收器*

*startReceiver(receiver, executors)*

*}*

*case RestartReceiver(receiver) =>*

*......*

*}*

1. 启动流数据接收器Receiver前，ReceiverSupervisor会向ReceiverTrackerEndpoint终端点发送RegisterReceiver注册消息，当注册成功后才继续进行数据接收器Receiver的启动。其中注册过程代码位于ReceiverSupervisor的startReceiver方法中，具体如下：

*def startReceiver(): Unit = synchronized {*

*try {*

*//先调用ReceiverSupervisorImpl类的onReceiverStart方法进行注册*

*//如果注册成功，则继续进行流数据接收器Receiver启动*

*if (onReceiverStart()) {*

*logInfo(s"Starting receiver $streamId")*

*receiverState = Started*

*receiver.onStart()*

*logInfo(s"Called receiver $streamId onStart")*

*} else {*

*// 如果Driver端的TrackerReceiver拒绝注册或注册失败，则停止流数据接收器*

*//并发送注销流数据接收器DeregisterReceiver消息*

*stop("Registered unsuccessfully because Driver refused to start receiver " + streamId, None)*

*}*

*}.....*

*}*

发送注册流数据接收器消息RegisterReceiver代码如下：

*override protected def onReceiverStart(): Boolean = {*

*val msg = RegisterReceiver(*

*streamId, receiver.getClass.getSimpleName, host, executorId, endpoint)*

*trackerEndpoint.askSync[Boolean](msg)*

*}*

1. 在流数据接收器Receiver接收数据的过程中，当保存完一个数据块时，作为数据转储的管理者ReceiverSupervisor会把数据块的元数据发送给ReceiverTrackerEndpoint终端点，ReceiverTracker再把这些信息转发给ReceivedBlockTracker，由它负责管理受到数据块的元信息。其中发送增加数据块消息代码位于ReceiverSupervisorImpl的pushAndReportBlocks方法中，具体代码如下：

*def pushAndReportBlock(*

*receivedBlock: ReceivedBlock,*

*metadataOption: Option[Any],*

*blockIdOption: Option[StreamBlockId]*

*) {*

*val blockId = blockIdOption.getOrElse(nextBlockId)*

*val time = System.currentTimeMillis*

*//调用ReceivedBlockHandler的storeBlock方法进行保存数据块*

*val blockStoreResult = receivedBlockHandler.storeBlock(blockId, receivedBlock)*

*logDebug(s"Pushed block $blockId in ${(System.currentTimeMillis - time)} ms")*

*val numRecords = blockStoreResult.numRecords*

*//把数据块的元信息发送给ReceiverTrackerEndpoint*

*val blockInfo = ReceivedBlockInfo(streamId, numRecords, metadataOption, blockStoreResult)*

*trackerEndpoint.askSync[Boolean](AddBlock(blockInfo))*

*logDebug(s"Reported block $blockId")*

*}*

1. 当Spark Streaming停止时，ReceiverTracker发送注销所有流数据接收器Receiver消息，ReceiverTrackerEndpoint终端点接到该消息会调用ReceiverTracker.stop方法注销。在停止操作过程中，ReceiverTracker会发送两次注销消息，发送消息之间的间隔10s，用于等待数据流接收器Receiver。

*def stop(graceful: Boolean): Unit = synchronized {*

*val isStarted: Boolean = isTrackerStarted*

*trackerState = Stopping*

*if (isStarted) {*

*if (!skipReceiverLaunch) {*

*// 第一次发送注销所有流数据接收器Receiver消息并等待10s*

*endpoint.askSync[Boolean](StopAllReceivers)*

*receiverJobExitLatch.await(10, TimeUnit.SECONDS)*

*//第二次发送注销所有流数据接收器Receiver消息，并返回注销结果*

*val receivers = endpoint.askSync[Seq[Int]](AllReceiverIds)*

*if (receivers.nonEmpty) {*

*logWarning("Not all of the receivers have deregistered, " + receivers)*

*} else {*

*logInfo("All of the receivers have deregistered successfully")*

*}*

*}*

*//停止endpoint*

*ssc.env.rpcEnv.stop(endpoint)*

*endpoint = null*

*}*

*receivedBlockTracker.stop()*

*logInfo("ReceiverTracker stopped")*

*trackerState = Stopped}*

## 2.2 Receiver分发

Spark Streaming中处理的数据一方面通过内部接口获取，另一方面来自于Kafka、Flume、Twitter、ZeroMQ以及TCP Sockets等外部系统源，获取后对这些输入的数据源进行Map、Reduce、Join和Window等操作，完成较为复杂的数据加工和处理，这些数据均抽象于Dstream，如下图：



直接继承于Dstream可以分为Dstream和Dstream等类别，而根据输入Dstream的来源不同可以分为ReceiverInputDStream(通过流接收器获取输入数据)、QueueInputStream、FileInputStream和DirectKafkaInputStream等方式。根据流接收器的类型又分为SocketInputStream、KafkaInputStream、FlumeInputStream和MQTTInputStream等类型。另外也支持通过PluggableInputStream自定义流接收器获取流数据。

不同的ReceiverInputDStream包含不同的流数据接收器，而这些接收器继承于Receiver，其类关系如下图所示：



在SparkStreaming启动过程中，ReceiverTracker会把流数据接收器Receiver分发到Executor上，在每个Executor上，由ReceiverSuperVisor启动对应的Receiver。

在Spark 1.5以后，StreamingContext会根据N个Receiver实例创建N个作业，各个作业只包含一个任务，并加入可插拔的Receiver分发策略，其结果如下图：



在Spark Streaming中每个Receiver都有一个作业来分发，而且对于这仅有一个任务只在第一次启动时才尝试启动Receiver，如果该任务失败，则不再尝试启动Receiver，对应的作业可以配置为完成状态。此时ReceiverTracker会新生成一个作业，在其他Executor尝试启动，直到成功为止，这样Reciever就不会受到任务失败上限而无法启动。通过这种方式，Spark Streaming中所有的Receiver总是保持活性，并不会收到Executor失败而停止。

可插拔的Receiver分发策略在ReceiverSchedulingPolicy类中定义，在Receiver分发之前会收集所有的InputStream包含的Receiver实例和Executor，然后调用该类中的scheduleReceivers方法计算每个Receiver对应的Executor。在该方法中，以轮询调度（Round-Robin）方式进行分配，首先对存在首选位置的Receiver进行处理，然后尽可能的把Receiver运行在首选位置机器运行Receiver个数最少的Executor中，接着对于没有首选位置的Receiver，则优先分配到运行Receiver个数最少的Executor中，分配完成后返回调度好的Executor列表：

*def scheduleReceivers(。、*

*receivers: Seq[Receiver[\_]],*

*executors: Seq[ExecutorCacheTaskLocation]): Map[Int, Seq[TaskLocation]] = {*

*//Executor格式host:port，获取冒号的前半部分信息，即IP地址或机器名*

*val hostToExecutors = executors.groupBy(\_.host)*

*val scheduledLocations = Array.fill(receivers.length)(new mutable.ArrayBuffer[TaskLocation])*

*//定义Executors，运行Receiver个数的哈希列表*

*val numReceiversOnExecutor = mutable.HashMap[ExecutorCacheTaskLocation, Int]()*

*executors.foreach(e => numReceiversOnExecutor(e) = 0)*

*//首先，如果Receiver存在首选位置，则把该首先位置放在候选调度Executor列表*

*for (i <- 0 until receivers.length) {*

*// Note: preferredLocation is host but executors are host\_executorId*

*receivers(i).preferredLocation.foreach { host =>*

*hostToExecutors.get(host) match {*

*//如果Receiver的首先位置在传入的Executor序列中，则在ScheduledExecutors的Receiver*

*//加入对应的Executor，在numReceiversOnExecutor对应的Executor的Receiver的个数加1*

*//需要注意的是，获取该首先位置机器运行Receiver个数最少的Executor*

*case Some(executorsOnHost) =>*

*val leastScheduledExecutor =*

*executorsOnHost.minBy(executor => numReceiversOnExecutor(executor))*

*scheduledLocations(i) += leastScheduledExecutor*

*numReceiversOnExecutor(leastScheduledExecutor) =*

*numReceiversOnExecutor(leastScheduledExecutor) + 1*

*//如果Receiver的首先位置不在传入的Executor序列中，则该Executor存在两种可能*

*//一种是还未启动完毕，另一种是ExecUtor已死或是所在主机不在集群中，对于这些情况*

*//则先把该主机加到ScheduledExecutors中*

*case None =>*

*scheduledLocations(i) += TaskLocation(host)*

*}*

*}*

*}*

*//对于那些不存在的首选位置的Receiver,及ScheduledExecutors数组对应为空的元素*

*//则在numReceiversOnExecutors获取Receiver计数最少的元素，把该Receiver放到*

*//其Executor进行运行，并更新numReceiversOnExecutor列表信息*

*for (scheduledLocationsForOneReceiver <- scheduledLocations.filter(\_.isEmpty)) {*

*// Select the executor that has the least receivers*

*val (leastScheduledExecutor, numReceivers) = numReceiversOnExecutor.minBy(\_.\_2)*

*scheduledLocationsForOneReceiver += leastScheduledExecutor*

*numReceiversOnExecutor(leastScheduledExecutor) = numReceivers + 1*

*}*

*//返回调度好的Executor列表，该列表和Receiver序列一一对应*

*receivers.map(\_.streamId).zip(scheduledLocations).toMap*

*}*

## 2.3 容错性

实时流处理系统需要长时间接收并处理数据，在这个过程中出现异常是难以避免的，需要流程系统具备高容错性。Spark Streaming在设置之初就考虑这种情况:一方面利用Spark自身的容错设计、存储级别（如设置存储级别为MEMORY\_AND\_DISK\_2）和RDD抽象设计能够处理集群中任何Worker节点的故障；另一方面由于Spark运行多种运行模式，其Driver端可能运行在Master节点或者集群中的任意节点，这样让Driver端具备容错能力是一个很大的挑战，但是由于Spark Streaming接收的数据是按照批进行存储和处理，这些批次数据的元数据可以通过执行检查点的方式定期写入到可靠的存储中，在Driver端重新启动中恢复这些状态。

虽然Spark Streaming在绝大多数下能够处理并恢复运行的状态和数据，但是对于接受数据存在于内存中存在丢失的风险。例如对于Kafka或者Flume等其他数据源，由于接收到的数据还只缓存在Executor内存中，尚未及时被处理，当Executor出现异常时会丢失这些数据；如果集群中处理独立运行模式、YARN运行模式或者Mesos运行模式，当Driver端失败时该Driver端所管理的Executor及内存中数据将终止，即使Driver端重新启动这些缓存的数据也不能被恢复。为了避免这些数据损失，Spark引进了预写日志（WrigeAheadLogs）功能。

预写日志通常被用于数据库和文件系统中，保证数据操作的持久性。预写日志通常是先将操作写入到一个持久可靠的日志文件中，然后才对数据施加该操作，当加入施加操作中出现异常，可以通过读取日志文件并重新施加该操作，从而恢复系统。当启动预写日志以后，所以收到的数据同时还保存到了容错文件系统的日志文件中，当Spark Streaming失败，这些接收到的数据也不会丢失。另外，接收数据的正确性只在数据被预写到日志以后接收器才会确认，已经缓存但还没有保存的数据可以在Driver重新启动之后由数据源再发送一次。这两个机制确保了零数据丢失，即所有的数据或者从日志中恢复，或者由数据源重发。

在一个Spark Streaming应用开始时（也就是Driver开始时），相关的StreamingContext(所有流功能的基础)使用SparkContext启动接收器成为长驻运行服务。这些接收器接收并保存流数据到Spark内存中以供处理。用户传送数据的生命周期如下图所示：



1. 接收数据，接收器将数据流分成一系列小块，存储到Executor内存或磁盘中。如果启用预写日志，数据同时还写入到容错文件系统的预写日志文件中。
2. 通知StreamingContext，接收块中的元数据（Metadata）被分到Driver的StreamingContext。这个元数据包括：定位其在Executor内存或磁盘中数据位置的块编号；块数据在日志中的偏移信息（如果启用了预写日志）。
3. 处理数据：每批数据的间隔，流上下文使用块信息产生弹性分布式集RDD和它们的作业Job，StreamingContext通过运行任务处理Executor内存或者磁盘中的数据块来执行作业。
4. 周期性地设置检查点，为了恢复的需要，流计算（StreamingContext提供的DStream）周期性的设置检查点，并保存到同一个容错文件系统中另外一组文件中。

当一个失败的Driver端重启时，会进行如下处理：



1. 恢复计算，使用检查点信息重启Driver端，重新构造上下文并重启接收器
2. 恢复元数据块，为了保证能够继续下去所必备的全部元数据块都被恢复
3. 未完成作业的重新形成，由于失败而没有处理完成的批处理，将使用恢复的元数据再次产生RDD和对应的作业
4. 读取保存在日志中的块数据，在这些作业执行时，块数据直接从预写入中读出，这些将恢复日志中可靠的保存所有必要的数据
5. 重发尚未确认的数据，失败时没有保存到日志中的缓存数据将由数据源再次发送。

因此确认预写日志和可靠的接收器，Spark Streaming就可以保证没有输入数据会由于Driver端的失败而丢失。但需要注意的是，在启用预写日志以后，数据接收吞吐率会轻微的降低，因为所有数据都写入容错文件系统，文件系统的写入吞吐率和用于数据复制的网络带宽，可能就是潜在的瓶颈。

# 3.Spark Streaming运行原理

## 3.1启动流处理引擎

在运行架构分析中，启动流处理引擎阶段需要初始化StreamingContext中的DstreamGraph和JobScheduler，进而启动JobScheduler中的ReceiverTracker和JobGenerator，其启动时序图如下所示：



1. 初始化StreamingContext

对于一个Spark Streaming应用程序，首先要做的事情是初始化StreamingContext，对Streaming初始化的可以有多个重载方法，在这里使用SparkConf的对象和Duration(批处理的时间间隔)作为参数传入StreamingContext的构造函数中，Checkpoint默认被设置为null，代码如下：

*def this(conf: SparkConf, batchDuration: Duration) = {*

*this(StreamingContext.createNewSparkContext(conf), null, batchDuration)*

*}*

与SparkContext类似，StreamingContext在初始化过程中也会进行成员变量的初始化，在这些成员变量中有DstreamGraph,JobScheduler和StreamingTab等。其中DstreamGraph跟RDD的有向无环图类似，包含Dstream之间相互依赖的有限无环图。JobScheduler的作用是定时查看DstreamGraph，然后根据流入的数据生成运行作业。StreamTab是在Spark Streaming的作业运行的时候，提供对流数据处理的监控。

1. 创建InputDStream

StreamingContext的socketTextStream方法生成具体的InputDStream，在socketTextStream方法有3个参数，其中hostname和port分别表示要连接服务端的主机名和端口号，而StorageLvel即数据的存储等级，默认是StorageLevel.MEMORY\_AND\_DISK\_SER\_2

*def socketStream[T: ClassTag](*

*hostname: String,*

*port: Int,*

*converter: (InputStream) => Iterator[T],*

*storageLevel: StorageLevel*

*): ReceiverInputDStream[T] = {*

*new SocketInputDStream[T](this, hostname, port, converter, storageLevel)*

*}*

继续跟踪socketStream方法，在其中创建了SocketInputStream，在Dstream类图中可以看到SocketInputDstream类继承自ReceiverInputDStream，而ReceiverInputDStream继承自InputDStream,InputDStream继承自Dstream。另外，在SocketInputDStream内部重新了getReceiver方法，该方法是用来生成接收器的，当Spark Streaming进行分发流接收器Receiver进行调用。getReceiver方法内部创建了一个SocketReceiver实例，并在该实例中启动线程。其中SocketInputStream代码如下：

*class SocketInputDStream[T: ClassTag](*

*\_ssc: StreamingContext,*

*host: String,*

*port: Int,*

*bytesToObjects: InputStream => Iterator[T],*

*storageLevel: StorageLevel*

*) extends ReceiverInputDStream[T](\_ssc) {*

*//创建SocketReceiver实例，在该实例中启动线程接收Socket数据去去2*

*def getReceiver(): Receiver[T] = {*

*new SocketReceiver(host, port, bytesToObjects, storageLevel)*

*}*

*}*

通过以上步骤创建InputDStream对象，接着就是对InputDStream进行flatMap、Map、reduceByKey和print等连续操作，类似RDD的转换操作。

1. 启动JobScheduler

创建完InputDStream后，调用StreamingContext的start方法进行Spark Streaming应用程序的启动，其中最重要的就是启动JobScheduler，启动过程中，实例化ReceiverTracker和JobGenerator，其中JobScheduler的启动代码如下：

*def start(): Unit = synchronized {*

*if (eventLoop != null) return //已经启动，则退出*

*eventLoop = new EventLoop[JobSchedulerEvent]("JobScheduler") {*

*override protected def onReceive(event: JobSchedulerEvent): Unit = processEvent(event)*

*override protected def onError(e: Throwable): Unit = reportError("Error in job scheduler", e)*

*}*

*eventLoop.start()*

*for {*

*inputDStream <- ssc.graph.getInputStreams*

*rateController <- inputDStream.rateController*

*} ssc.addStreamingListener(rateController)*

*//启动Spark Streaming的消息总线*

*listenerBus.start()*

*receiverTracker = new ReceiverTracker(ssc)*

*inputInfoTracker = new InputInfoTracker(ssc)*

*val executorAllocClient: ExecutorAllocationClient = ssc.sparkContext.schedulerBackend match {*

*case b: ExecutorAllocationClient => b.asInstanceOf[ExecutorAllocationClient]*

*case \_ => null*

*}*

*//启动ReceiverTracker和JobGenerator*

*receiverTracker.start()*

*jobGenerator.start()*

*executorAllocationManager.foreach(\_.start())*

*logInfo("Started JobScheduler")*

*}*

1. 启动JobGenerator

启动JobGenerator需要判断是否第一次运行，如果不是第一次运行需要进行上次检查点的恢复，如果是第一次运行，则调用了JobGenerator和startFirstTime方法。在该方法中初始化定时器的开启时间，并启动DstreamGraph和定时器timer。

## 3.2 接收及存储流数据

1）启动ReceiverTracker

启动ReceiverTracker先调用ReceiverTracker.launchReceivers方法，该方法会向ReceiverT

rackerEndpoint终端点发送分支，并启动所有流数据接收器的消息，在ReceiverTracker

.startReceiver的方法中进行所有流数据接收器Receiver的分发并启动，其中launchReceivers代码如下：

*private def launchReceivers(): Unit = {*

*//获取所有InputDStream中定义的流数据接收器*

*val receivers = receiverInputStreams.map { nis =>*

*val rcvr = nis.getReceiver()*

*rcvr.setReceiverId(nis.id)*

*rcvr*

*}*

*runDummySparkJob()*

*//向终端点ReceiverTrackerEndpoint发送分发并启动所有流数据接收器的消息*

*endpoint.send(StartAllReceivers(receivers))*

*}*

ReceiverTracker启动过程时序图调用如下所示：



分发到Executor的流数据接收器由ReceiverTrackerEndpoint.startReceiver方法进行启动，在该方法中开启了一个ReceiverSuperVisor对象来管理该流数据接收器，其代码如下：

*private def startReceiver(*

*receiver: Receiver[\_],*

*scheduledLocations: Seq[TaskLocation]): Unit = {*

*......*

*val startReceiverFunc: Iterator[Receiver[\_]] => Unit =*

*(iterator: Iterator[Receiver[\_]]) => {*

*if (!iterator.hasNext) {*

*throw new SparkException(*

*"Could not start receiver as object not found.")*

*}*

*if (TaskContext.get().attemptNumber() == 0) {*

*val receiver = iterator.next()*

*assert(iterator.hasNext == false)*

*//创建流数据接收器管理器，用于监管该流数据接收器*

*val supervisor = new ReceiverSupervisorImpl(*

*receiver, SparkEnv.get, serializableHadoopConf.value, checkpointDirOption)*

*supervisor.start()*

*supervisor.awaitTermination()*

*} else {*

*// It's restarted by TaskScheduler, but we want to reschedule it again. So exit it.*

*}*

*}}*

在ReceiverSupervisor的start方法中调用了其自身onStart和startReceiver两个方法，在onStart方法中启动了BlockGenerator，而在startReceiver方法中完成了流数据接收器的注册和启动。

2.启动流数据接收器并接收数据

在ReceiverSupervisor.startReceiver方法进行流数据接收器的启动，先调用ReceiverSupervisorImpl的onReceiverStart方法向ReceiverTrackerEndpoint终端点发送RegisterReceiver注册消息，如果注册成功了，则调用SocketReceiver.onStart方法开始接收数据。

在SocketReceiver.onStart中接收数据调用了receive方法，该方法中通过监听Socket端口传递过来的流数据，并把接收到的数据转化为对象进行保存：

*def receive() {*

*try {*

*//根据机器名和端口建立Socket连接*

*val iterator = bytesToObjects(socket.getInputStream())*

*while(!isStopped && iterator.hasNext) {*

*store(iterator.next())*

*}*

*if (!isStopped()) {*

*restart("Socket data stream had no more data")*

*} else {*

*logInfo("Stopped receiving")*

*}*

*} catch {*

*case NonFatal(e) =>*

*logWarning("Error receiving data", e)*

*restart("Error receiving data", e)*

*} finally {*

*//关闭socket连接*

*onStop()*

*}*

*}*

*}*

3.启动BlockGenerator并生成数据块

在ReceiverSupervisorImpl.onStart方法中调用BlockGenerator.start启动BlockGenerator。而BlockGenerator.start方法主要完成两件事情：启动一个数据生成定时器，将当前currentBuffer缓存中的数据按照用户在Spark Streaing应用程序里定义的批处理时间间隔封装了Block的数据块，然后存放到BlockGenerator的blocksForPushing队列中；启动一个BlockPushingThread线程，不断的将BlockForPush队列中的数据块传递给BlockManager。

*private val blockIntervalTimer =*

*new RecurringTimer(clock, blockIntervalMs, updateCurrentBuffer, "BlockGenerator")*

*private val blockQueueSize = conf.getInt("spark.streaming.blockQueueSize", 10)*

*private val blocksForPushing = new ArrayBlockingQueue[Block](blockQueueSize)*

*private val blockPushingThread = new Thread() { override def run() { keepPushingBlocks() } }*

*@volatile private var currentBuffer = new ArrayBuffer[Any]*

*@volatile private var state = Initialized*

*/\*\* Start block generating and pushing threads. \*/*

*def start(): Unit = synchronized {*

*if (state == Initialized) {*

*state = Active*

*//开启一个定时器，定期的把缓存中的数据封装成数据块*

*blockIntervalTimer.start()*

*//开始一个线程，不断将封装好的数据推送给BlockManager*

*blockPushingThread.start()*

*logInfo("Started BlockGenerator")*

*} else {*

*throw new SparkException(*

*s"Cannot start BlockGenerator as its not in the Initialized state [state = $state]")*

*}*}

根据blockIntervalTimer.start方法中，定时器的执行流程start(getStartTime())，而start方法中开启一个线程来执行RecurringTimer.loop方法，代码如下：

*private def triggerActionForNextInterval(): Unit = {*

*clock.waitTillTime(nextTime)*

*//执行回调函数，该函数为updateCurrentBuffer*

*callback(nextTime)*

*prevTime = nextTime*

*nextTime += period*

*logDebug("Callback for " + name + " called at time " + prevTime)*

*}*

在loop方法中是数据块生成定时器真正处理的工作，定时执行回调函数，其为初始化数据块生成定时器传入的updateCurrentBuffer参数。在该回到函数中，先把内存的数据currentBuffer赋值给newBlockBuffer，然后把newBlockBuffer封装成一个数据块，最后把整个数据块放进blockForPushing队列中，该队列默认情况下数目为10个。其中BlockGenerator.

updateCurrentBuffer代码如下：

*private def updateCurrentBuffer(time: Long): Unit = {*

*try {*

*var newBlock: Block = null*

*synchronized {*

*if (currentBuffer.nonEmpty) {*

*val newBlockBuffer = currentBuffer*

*currentBuffer = new ArrayBuffer[Any]*

*val blockId = StreamBlockId(receiverId, time - blockIntervalMs)*

*listener.onGenerateBlock(blockId)*

*newBlock = new Block(blockId, newBlockBuffer)*

*}*

*}*

*if (newBlock != null) {*

*blocksForPushing.put(newBlock) // put is blocking when queue is full*

*}*

*} catch {}*

*}*

在加入currentBuffer数组会先由RateLimiter检查一下速率，是否加入的频率太高等。如果太高，需要阻塞等到下一秒再开始添加。最高频率由：

*spark.streaing.receiver.maxRate*

控制的，是单个流数据接收器每秒运行添加的条数。控制这个速率，控制整个SparkStreaming系统每个批处理需要处理的最大量。

BlockGenerator生成的数据块使用keepPushingBlock方法传递给BlockManager，该操作由pushBlock方法进行具体失败，而在pushBlock中继续调用listener.onPushBlock，其中listener实在ReceiverSupervisorImpl中初始化BlockGenerator时传递给它的参数BlockGneratorListener对象。在BlockGeneratorListenr.onPushBlock方法里会继续调用ReceiversupervisorImpl.pushArrayBuffer方法，在pushArrayBuffer方法中则调用该类下的pushAndReportBlocks方发送AddBlock消息给ReceiverTracker，进行数据块的报错。其中keepPushingBlocks代码如下：

*private def keepPushingBlocks() {*

*def areBlocksBeingGenerated: Boolean = synchronized {*

*state != StoppedGeneratingBlocks*

*}*

*try {*

*// 不断从blocksForPushing队列中抽取数据块，时间间隔10ms*

*while (areBlocksBeingGenerated) {*

*Option(blocksForPushing.poll(10, TimeUnit.MILLISECONDS)) match {*

*case Some(block) => pushBlock(block)*

*case None =>*

*}*

*}*

*//在退出之前会判断blocksForPushing是否为空，不为空，则会在退出之前把*

*//剩下的数据块一起输出*

*while (!blocksForPushing.isEmpty) {*

*val block = blocksForPushing.take()*

*pushBlock(block)*

*}*

*} catch {}*

*}*

1. 数据存储

作为流数据接收器调用Receiver.store方法进行数据存储，该方法中有多个重载方法，如果数据量很小，则攒多条数据成数据块再进行块存储；如果数据量太大，则直接进行块存储。对于需要攒数据块的操作，在Receiver.store方法中调用ReceiverSupervisorImpl的pushSingle方法进行处理。在pushSingle方法中，通过调用BlockGenerator.addData把数据保存到内存currentBuffer中，这些内存中的数据被数据块生成定时器封装加入队列并调用ReceiverSupervisor的pushArrayBuffer方法进行处理。而对于直接存储的操作，则调用ReceiverSupervisor的pushIterator或pushBytes方法进行处理。

这两种情况下均是调用pushAndReportBlock方法进行数据存储，该方法一方面会调用ReceivedBlockHandler的storeBlock方法保存数据并根据配置进行预写日志，另一方面会处理好的数据块元信息发送给ReceiverTrackerEndpoint终端点，ReceiverTracker再把这些信息转发给ReceivedBlockTracker，由它负责管理收到数据块的元数据。

其中ReceivedBlockHandler根据SparkEnv中配置项的实际值来决定，如果设置了预写日志，即实例化WriteAheadLogBasedBlockHandler；如果未设置预写日志，则实例化BlockManagerBasedBlockHandler，其在ReceiverSupervisorImpl类中代码如下：

*private val receivedBlockHandler: ReceivedBlockHandler = {*

*if (WriteAheadLogUtils.enableReceiverLog(env.conf)) {*

*if (checkpointDirOption.isEmpty) {*

*throw new SparkException(*

*"Cannot enable receiver write-ahead log without checkpoint directory set. " +*

*"Please use streamingContext.checkpoint() to set the checkpoint directory. " +*

*"See documentation for more details.")*

*}*

*new WriteAheadLogBasedBlockHandler(env.blockManager, env.serializerManager, receiver.streamId,*

*receiver.storageLevel, env.conf, hadoopConf, checkpointDirOption.get)*

*} else {*

*new BlockManagerBasedBlockHandler(env.blockManager, receiver.storageLevel)*

*}*

*}*

其中WriteAheadLogBasedBlockHandler.storeBlock方法，会使用BlockManager的putBytes方法，把数据保存到Executor的内存或者磁盘中，然后把数据写入到预写日志文件中，最后当这两个写入过程完成后返回写入数据块的元数据：

*def storeBlock(blockId: StreamBlockId, block: ReceivedBlock): ReceivedBlockStoreResult = {*

*......*

*//使用BlockManager，把数据保存到内存或者磁盘*

*val storeInBlockManagerFuture = Future {*

*val putSucceeded = blockManager.putBytes(*

*blockId,*

*serializedBlock,*

*effectiveStorageLevel,*

*tellMaster = true)*

*if (!putSucceeded) {*

*throw new SparkException(*

*s"Could not store $blockId to block manager with storage level $storageLevel")*

*}*

*}*

*// 调用WriteAheadLogRecordHandle中write方法在预写日志文件中写入数据*

*val storeInWriteAheadLogFuture = Future {*

*writeAheadLog.write(serializedBlock.toByteBuffer, clock.getTimeMillis())*

*}*

*// 等待写入日志和数据完成，完成后返回存储块数据的元数据*

*val combinedFuture = storeInBlockManagerFuture.zip(storeInWriteAheadLogFuture).map(\_.\_2)*

*val walRecordHandle = ThreadUtils.awaitResult(combinedFuture, blockStoreTimeout)*

*WriteAheadLogBasedStoreResult(blockId, numRecords, walRecordHandle)*

*}*

数据存储类时序图如下所示：



## 3.3 数据处理

Spark Streaming在生成SocketInputStream对象后，会进行flatMap、map、reduceByKey和print等一系列操作，其中print的源码如下：

*def print(num: Int): Unit = ssc.withScope {*

*def foreachFunc: (RDD[T], Time) => Unit = {*

*(rdd: RDD[T], time: Time) => {*

*val firstNum = rdd.take(num + 1)*

*// scalastyle:off println*

*println("-------------------------------------------")*

*println(s"Time: $time")*

*println("-------------------------------------------")*

*firstNum.take(num).foreach(println)*

*if (firstNum.length > num) println("...")*

*println()*

*// scalastyle:on println*

*}*

*}*

*New ForEachDStream(this,*

*context.sparkContext.clean(foreachFunc, false), displayInnerRDDOps).register()*

*}*

print方法是DstreamGraph的最后一个操作，像RDD的行动操作。在这个方法中，生成foreachRDD，并定义一个用于作用它的函数foreachFunc，最后调用register方法里，调用DstreamGraph的addOutputStream方法，可以看到初始化的ForEachDStream已经添加DstreamGraph的一个ArrayBuffer[Dstream[\_]]中。

在启动JobGenerator中启动了作业生成器的定时器timer，其定义如下：

*val clock = {*

*val clockClass = ssc.sc.conf.get(*

*"spark.streaming.clock", "org.apache.spark.util.SystemClock")*

*try {*

*Utils.classForName(clockClass).newInstance().asInstanceOf[Clock]*

*} catch {……*

*}*

*private val timer = new RecurringTimer(clock, ssc.graph.batchDuration.milliseconds,*

*longTime => eventLoop.post(GenerateJobs(new Time(longTime))), "JobGenerator")*

在定时器timer中会定时的调用processEvent方法，对于接收到的GenerateJobs消息会调用JobGenerator.generateJobs方法继续处理：

*private def generateJobs(time: Time) {*

*ssc.sparkContext.setLocalProperty(RDD.CHECKPOINT\_ALL\_MARKED\_ANCESTORS, "true")*

*Try {*

*jobScheduler.receiverTracker.allocateBlocksToBatch(time) // allocate received blocks to batch*

*graph.generateJobs(time) // generate jobs using allocated block*

*} match {*

*case Success(jobs) =>*

*val streamIdToInputInfos = jobScheduler.inputInfoTracker.getInfo(time)*

*jobScheduler.submitJobSet(JobSet(time, jobs, streamIdToInputInfos))*

*case Failure(e) =>*

*jobScheduler.reportError("Error generating jobs for time " + time, e)*

*PythonDStream.stopStreamingContextIfPythonProcessIsDead(e)*

*}*

*eventLoop.post(DoCheckpoint(time, clearCheckpointDataLater = false))*

*}*

在JobGeneratorJobs的generatorJobs方法中，主要进行5个步骤：

1. 要求ReceiverTracker将目前已经收到的数据进行一次提交，即将上次批处理切分后未处理的数据切分到本次批处理中。ReceiverTracker的allocatBlocksToBatch方法继续调用ReceivedBlockTracker的allocateBlocksToBatch方法对本次批处理进行分配数据块，处理之前把批处理时间和对应数据块的元信息记录到预写日志文件中，具体代码如下：

*def allocateBlocksToBatch(batchTime: Time): Unit = synchronized {*

*if (lastAllocatedBatchTime == null || batchTime > lastAllocatedBatchTime) {*

*val streamIdToBlocks = streamIds.map { streamId =>*

*(streamId, getReceivedBlockQueue(streamId).dequeueAll(x => true))*

*}.toMap*

*val allocatedBlocks = AllocatedBlocks(streamIdToBlocks)*

*if (writeToLog(BatchAllocationEvent(batchTime, allocatedBlocks))) {*

*timeToAllocatedBlocks.put(batchTime, allocatedBlocks)*

*lastAllocatedBatchTime = batchTime*

*} else {*

*logInfo(s"Possibly processed batch $batchTime needs to be processed again in WAL recovery")*

*}*

*} else {*

*logInfo(s"Possibly processed batch $batchTime needs to be processed again in WAL recovery")*

*}*

*}*

2)调用DstreamGraph的generateJobs方法生成作业，在这个方法内部会通过outputStream的generateJob方法生成作业

*def generateJobs(time: Time): Seq[Job] = {*

*val jobs = this.synchronized {*

*outputStreams.flatMap { outputStream =>*

*val jobOption = outputStream.generateJob(time)*

*jobOption.foreach(\_.setCallSite(outputStream.creationSite))*

*jobOption*

*}*

*}*

*logDebug("Generated " + jobs.length + " jobs for time " + time)*

*jobs*

*}*

这里的outputStream是我们前面提到的ForEachStream，ForEachStream的generateJob方法会调用Dstream的getOrCompute方法生成RDD，然后再定义一个作用于作业的jobFunc，最后会初始化一个作业，并把定义好的jobFunc函数作为参数传给作业。

*private[streaming] def generateJob(time: Time): Option[Job] = {*

*getOrCompute(time) match {*

*case Some(rdd) =>*

*val jobFunc = () => {*

*val emptyFunc = { (iterator: Iterator[T]) => {} }*

*context.sparkContext.runJob(rdd, emptyFunc)*

*}*

*Some(new Job(time, jobFunc))*

*case None => None*

*}*

继续看Dstream的getOrCompute方法看他们如何生成RDD:

*private[streaming] final def getOrCompute(time: Time): Option[RDD[T]] = {*

*generatedRDDs.get(time).orElse {*

*//每个DStream在自身实现的compute中会进行重写*

*if (isTimeValid(time)) {*

*val rddOption = createRDDWithLocalProperties(time, displayInnerRDDOps = false) {*

*SparkHadoopWriterUtils.disableOutputSpecValidation.withValue(true) {*

*compute(time)*

*}*

*}*

*rddOption.foreach { case newRDD =>*

*// Register the generated RDD for caching and checkpointing*

*if (storageLevel != StorageLevel.NONE) {*

*newRDD.persist(storageLevel)*

*logDebug(s"Persisting RDD ${newRDD.id} for time $time to $storageLevel")*

*}*

*if (checkpointDuration != null && (time - zeroTime).isMultipleOf(checkpointDuration)) {*

*newRDD.checkpoint()*

*logInfo(s"Marking RDD ${newRDD.id} for time $time for checkpointing")*

*}*

*//把新生成的RDD保存到generateRDDS的哈希map中*

*generatedRDDs.put(time, newRDD)*

*}*

*rddOption*

*} else {*

*None*

*}*

*}*

*}*

3）获取接收到的Block信息，通过调用InputInfoTracker的getInfo方法把Receivertracker中接收到的数据块元数据，保存到batchTimeToInputInfos中的hashMap中，然后以参数的形式传递给JobSet：

*def getInfo(batchTime: Time): Map[Int, StreamInputInfo] = synchronized {*

*val inputInfos = batchTimeToInputInfos.get(batchTime)*

*// Convert mutable HashMap to immutable Map for the caller*

*inputInfos.map(\_.toMap).getOrElse(Map[Int, StreamInputInfo]())*

*}*

4）调用JobScheduler的submitJobSet方法提交作业，在这个方法中，最重要的代码是jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))，遍历JobSet里所有的作业，然后通过jobExecutor这个线程池把所有的作业进行提交。JobExecutor作为JobScheduler的成员变量进行初始化。JobHandler类主要的两件事情：在作业运行前后分别发送JobStarted和JobComplted消息给JobScheduler：进行作业的运行。

*def submitJobSet(jobSet: JobSet) {*

*if (jobSet.jobs.isEmpty) {*

*logInfo("No jobs added for time " + jobSet.time)*

*} else {*

*listenerBus.post(StreamingListenerBatchSubmitted(jobSet.toBatchInfo))*

*jobSets.put(jobSet.time, jobSet)*

*jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))*

*logInfo("Added jobs for time " + jobSet.time)*

*}*

*}*

5）提交完作业后，发送一个DoCheckpoint消息给JobGenerator，然后调用JobGenerator的doCheckpoint进行检查点操作，该操作是异步的，不用等检查点真正写完成即可返回，数据处理的类时序图如下所示：

