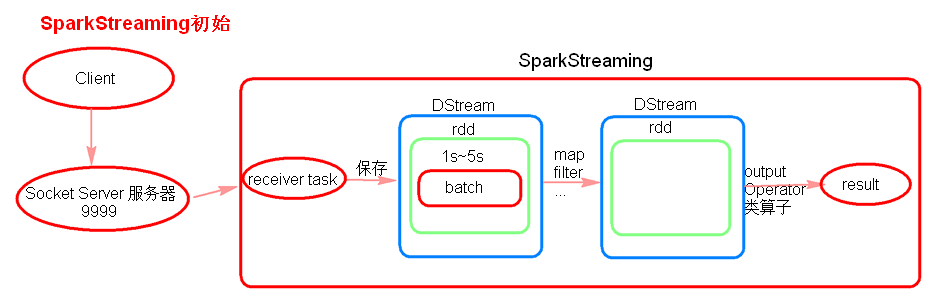
# Spark day06

1. SparkStreaming简介

SparkStreaming是流式处理框架，是Spark API的扩展，支持可扩展、高吞吐量、容错的实时数据流处理，实时数据的来源可以是：Kafka, Flume, Twitter, ZeroMQ或者TCP sockets，并且可以使用高级功能的复杂算子来处理流数据。例如：map,reduce,join,window 。最终，处理后的数据可以存放在文件系统，数据库等，方便实时展现。

1. SparkStreaming与Storm的区别
2. Storm是纯实时的流式处理框架，SparkStreaming是准实时的处理框架（微批处理）。因为微批处理，SparkStreaming的吞吐量比Storm要高。
3. Storm 的事务机制要比SparkStreaming的要完善。
4. Storm支持动态资源调度。(spark1.2开始和之后也支持)
5. SparkStreaming擅长复杂的业务处理，Storm不擅长复杂的业务处理，擅长简单的汇总型计算。
6. SparkStreaming初始
7. SparkStreaming初始理解



注意:

* receiver task是7\*24小时一直在执行，一直接受数据，将一段时间内接收来的数据保存到batch中。假设batchInterval为5s,那么会将接收来的数据每隔5秒封装到一个batch中，batch没有分布式计算特性，这一个batch的数据又被封装到一个RDD中，RDD最终封装到一个DStream中。

例如：假设batchInterval为5秒，每隔5秒通过SparkStreaming将得到一个DStream,在第6秒的时候计算这5秒的数据，假设执行任务的时间是3秒,那么第6~9秒一边在接收数据，一边在计算任务，9~10秒只是在接收数据。然后在第11秒的时候重复上面的操作。

* 如果job执行的时间大于batchInterval会有什么样的问题？

如果接受过来的数据设置的级别是仅内存，接收来的数据会越堆积越多，最后可能会导致OOM（如果设置StorageLevel包含disk, 则内存存放不下的数据会溢写至disk, 加大延迟 ）。

1. SparkStreaming代码

代码注意事项：

* 启动socket server 服务器：nc –lk 9999
* receiver模式下接受数据，local的模拟线程必须大于等于2，一个线程用来receiver用来接受数据，另一个线程用来执行job。
* Durations时间设置就是我们能接收的延迟度。这个需要根据集群的资源情况以及任务的执行情况来调节。
* 创建JavaStreamingContext有两种方式（SparkConf,SparkContext）
* 所有的代码逻辑完成后要有一个output operation类算子。
* JavaStreamingContext.start() Streaming框架启动后不能再次添加业务逻辑。
* JavaStreamingContext.stop() 无参的stop方法将SparkContext一同关闭，stop(false)，不会关闭SparkContext。
* JavaStreamingContext.stop()停止之后不能再调用start。

|  |
| --- |
| SparkConf conf = **new** SparkConf().setMaster("local[2]").setAppName("WordCountOnline");  /\*\*  \* 在创建streaminContext的时候 设置batch Interval  \*/  JavaStreamingContext jsc = **new** JavaStreamingContext(conf, Durations.*seconds*(5));  JavaReceiverInputDStream<String> lines = jsc.socketTextStream("node5", 9999);  JavaDStream<String> words = lines.flatMap(**new** FlatMapFunction<String, String>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Iterable<String> call(String s) {  **return** Arrays.*asList*(s.split(" "));  }  });  JavaPairDStream<String, Integer> ones = words.mapToPair(**new** PairFunction<String, String, Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Tuple2<String, Integer> call(String s) {  **return** **new** Tuple2<String, Integer>(s, 1);  }  });  JavaPairDStream<String, Integer> counts = ones.reduceByKey(**new** Function2<Integer, Integer, Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Integer call(Integer i1, Integer i2) {  **return** i1 + i2;  }  });    //outputoperator类的算子  counts.print();    jsc.start();  //等待spark程序被终止  jsc.awaitTermination();  jsc.stop(**false**); |

1. SparkStreaming算子操作
2. foreachRDD

* output operation算子,必须对抽取出来的RDD执行action类算子，代码才能执行。

1. transform

* transformation类算子
* 可以通过transform算子，对Dstream做RDD到RDD的任意操作。

1. updateStateByKey

* transformation算子
* updateStateByKey作用：

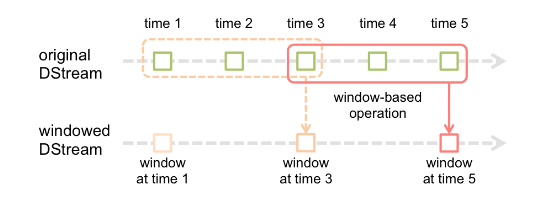
1. 为SparkStreaming中每一个Key维护一份state状态，state类型可以是任意类型的，可以是一个自定义的对象，更新函数也可以是自定义的。
2. 通过更新函数对该key的状态不断更新，对于每个新的batch而言，SparkStreaming会在使用updateStateByKey的时候为已经存在的key进行state的状态更新。

* 使用到updateStateByKey要开启checkpoint机制和功能。
* 多久会将内存中的数据写入到磁盘一份？

如果batchInterval设置的时间小于10秒，那么10秒写入磁盘一份。如果batchInterval设置的时间大于10秒，那么就会batchInterval时间间隔写入磁盘一份。

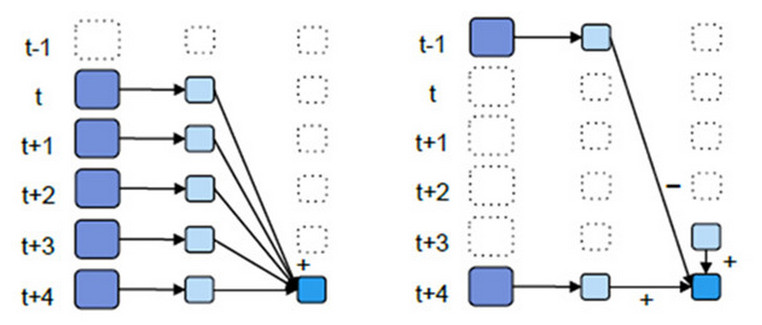
1. 窗口操作

* 窗口操作理解图：



假设每隔5s 1个batch,上图中窗口长度为15s，窗口滑动间隔10s。

* 窗口长度和滑动间隔必须是batchInterval的整数倍。如果不是整数倍会检测报错。
* 优化后的window窗口操作示意图：



* 优化后的window操作要保存状态所以要设置checkpoint路径，没有优化的window操作可以不设置checkpoint路径。

1. Driver HA（Standalone或者Mesos）

因为SparkStreaming是7\*24小时运行，Driver只是一个简单的进程，有可能挂掉，所以实现Driver的HA就有必要（如果使用的Client模式就无法实现Driver HA ，这里针对的是cluster模式）。Yarn平台cluster模式提交任务，AM(AplicationMaster)相当于Driver，如果挂掉会自动启动AM。这里所说的DriverHA针对的是Spark standalone和Mesos资源调度的情况下。实现Driver的高可用有两个步骤:

第一：提交任务层面，在提交任务的时候加上选项 --supervise,当Driver挂掉的时候会自动重启Driver。

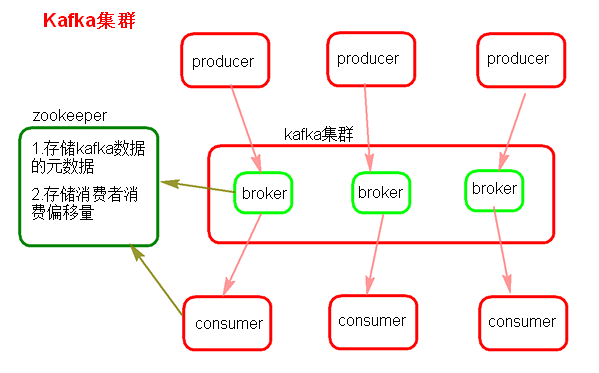
第二：代码层面，使用JavaStreamingContext.getOrCreate（checkpoint路径，JavaStreamingContextFactory）

* Driver中元数据包括：

1. 创建应用程序的配置信息。
2. DStream的操作逻辑。
3. job中没有完成的批次数据，也就是job的执行进度。
4. Kafka
5. kafka是什么？使用场景？

kafka是一个高吞吐的分布式消息队列系统。特点是生产者消费者模式，先进先出（FIFO）保证顺序，自己不丢数据，默认每隔7天清理数据。消息列队常见场景：系统之间解耦合、峰值压力缓冲、异步通信。

1. kafka生产消息、存储消息、消费消息



Kafka架构是由producer（消息生产者）、consumer（消息消费者）、borker(kafka集群的server，负责处理消息读、写请求，存储消息，在kafka cluster这一层这里，其实里面是有很多个broker)、topic（消息队列/分类相当于队列，里面有生产者和消费者模型）、zookeeper(元数据信息存在zookeeper中，包括：存储消费偏移量，topic话题信息，partition信息) 这些部分组成。

kafka里面的消息是有topic来组织的，简单的我们可以想象为一个队列，一个队列就是一个topic，然后它把每个topic又分为很多个partition，这个是为了做并行的，在每个partition内部消息强有序，相当于有序的队列，其中每个消息都有个序号offset，比如0到12，从前面读往后面写。一个partition对应一个broker，一个broker可以管多个partition，比如说，topic有6个partition，有两个broker，那每个broker就管3个partition。这个partition可以很简单想象为一个文件，当数据发过来的时候它就往这个partition上面append，追加就行，消息不经过内存缓冲，直接写入文件，kafka和很多消息系统不一样，很多消息系统是消费完了我就把它删掉，而kafka是根据时间策略删除，而不是消费完就删除，在kafka里面没有一个消费完这么个概念，只有过期这样一个概念。

producer自己决定往哪个partition里面去写，这里有一些的策略，譬如hash。consumer自己维护消费到哪个offset，每个consumer都有对应的group，group内是queue消费模型（各个consumer消费不同的partition，因此一个消息在group内只消费一次），group间是publish-subscribe消费模型，各个group各自独立消费，互不影响，因此一个消息在被每个group消费一次。

1. kafka的特点

* 系统的特点：生产者消费者模型，FIFO

Partition内部是FIFO的，partition之间呢不是FIFO的，当然我们可以把topic设为一个partition，这样就是严格的FIFO。

* 高性能：单节点支持上千个客户端，百MB/s吞吐，接近网卡的极限
* 持久性：消息直接持久化在普通磁盘上且性能好

直接写到磁盘中去，就是直接append到磁盘里去，这样的好处是直接持久化，数据不会丢失，第二个好处是顺序写，然后消费数据也是顺序的读，所以持久化的同时还能保证顺序，比较好，因为磁盘顺序读比较好。

* 分布式：数据副本冗余、流量负载均衡、可扩展

分布式，数据副本，也就是同一份数据可以到不同的broker上面去，也就是当一份数据，磁盘坏掉的时候，数据不会丢失，比如3个副本，就是在3个机器磁盘都坏掉的情况下数据才会丢，在大量使用情况下看这样是非常好的，负载均衡，可扩展，在线扩展，不需要停服务。

* 很灵活：消息长时间持久化+Client维护消费状态

消费方式非常灵活，第一原因是消息持久化时间跨度比较长，一天或者一星期等，第二消费状态自己维护消费到哪个地方了可以自定义消费偏移量。

1. kafka集群搭建
2. 上传kafka\_2.10-0.8.2.2.tgz包到三个不同节点上，解压。
3. 配置../ kafka\_2.10-0.8.2.2/config/server.properties文件

节点编号：（不同节点按0,1,2,3整数来配置）



真实数据存储位置：



zookeeper的节点：



1. 启动zookeeper集群。
2. 三个节点上，启动kafka:

|  |
| --- |
| bin/kafka-server-start.sh config/server.properties |

最好使用自己写的脚本启动，将启动命令写入到一个文件：

|  |
| --- |
| nohup bin/kafka-server-start.sh config/server.properties > kafka.log 2>&1 &  脚本附件：  **（放在与bin同一级别下，注意创建后要修改权限：chmod 755 startkafka.sh）** |

1. 相关命令：

创建topic：

|  |
| --- |
| **./kafka-topics.sh** **--zookeeper** node3:2181,node4:2181,node5:2181 **--create --topic** topic2017 **--partitions** 3 **--replication-factor** 3 |

用一台节点控制台来当kafka的生产者：

|  |
| --- |
| **./kafka-console-producer.sh**  **--topic** topic2017  **--broker-list** node1:9092,node2:9092,node3:9092 |

用另一台节点控制台来当kafka的消费者：

|  |
| --- |
| **./kafka-console-consumer.sh** **--zookeeper** node3:2181,node4:2181,node5:2181 **--topic** topic2017 |

查看kafka中topic列表:

|  |
| --- |
| **./kafka-topics.sh --list --zookeeper** node3:2181,node4:2181,node5:2181 |

查看kafka中topic的描述：

|  |
| --- |
| **./kafka-topics.sh --describe --zookeeper** node3:2181,node4:2181,node5:2181 --topic topic2017    注意：ISR是检查数据的完整性有哪些个节点。 |

查看zookeeper中topic相关信息：

|  |
| --- |
| 启动zookeeper客户端：  ./zkCli.sh  查看topic相关信息：  **ls** /brokers/topics/  查看消费者相关信息：  **ls** /consumers |

1. 删除kafka中的数据。
   1. ：在kafka集群中删除topic，当前topic被标记成删除。

|  |
| --- |
| ./kafka-topics.sh --zookeeper node3:2181,node4:2181,node5:2181 --delete --topic t1205 |

在每台broker节点上删除当前这个topic对应的真实数据。

* 1. ：进入zookeeper客户端，删除topic信息

|  |
| --- |
| rmr /brokers/topics/t1205 |

* 1. ：删除zookeeper中被标记为删除的topic信息

|  |
| --- |
| rmr /admin/delete\_topics/t1205 |

1. kafka的leader的均衡机制

当一个broker停止或者crashes时，所有本来将它作为leader的分区将会把leader转移到其他broker上去，极端情况下，会导致同一个leader管理多个分区，导致负载不均衡，同时当这个broker重启时，如果这个broker不再是任何分区的leader,kafka的client也不会从这个broker来读取消息，从而导致资源的浪费。

kafka中有一个被称为优先副本（preferred replicas）的概念。如果一个分区有3个副本，且这3个副本的优先级别分别为0,1,2，根据优先副本的概念，0会作为leader 。当0节点的broker挂掉时，会启动1这个节点broker当做leader。当0节点的broker再次启动后，会自动恢复为此partition的leader。不会导致负载不均衡和资源浪费，这就是leader的均衡机制。

在配置文件conf/ server.properties中配置开启（默认就是开启）：

|  |
| --- |
| auto.leader.rebalance.enable true |

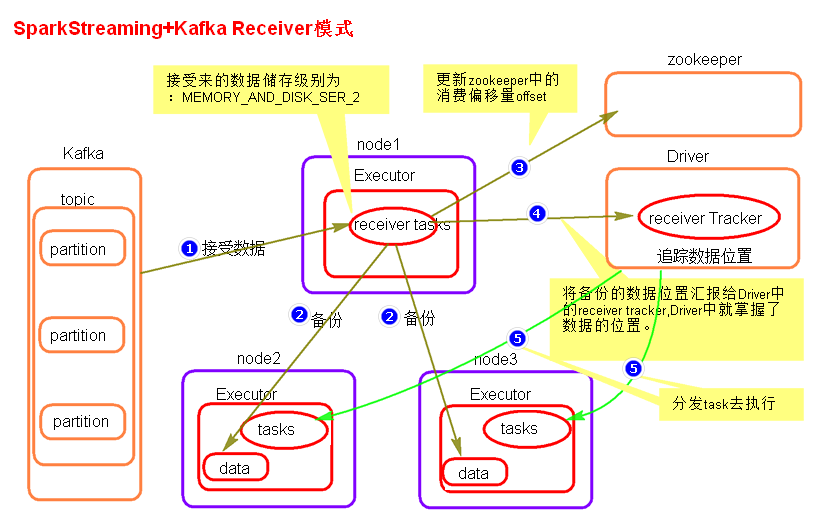
1. kafka 0.11版本改变

kafka 0.8.2版本消费者offset存储在zookeeper中，对于zookeeper而言每次写操作代价是很昂贵的，而且zookeeper集群是不能扩展写能力。kafka 0.11版本默认使用新的消费者api ,消费者offset会更新到一个kafka自带的topic【\_\_consumer\_offsets】中。以消费者组groupid 为单位，可以查询每个组的消费topic情况：

|  |
| --- |
| #查看所有消费者组  ./kafka-consumer-groups.sh --bootstrap-server c7node1:9092,c7node2:9092,c7node3:9092 --list  #查看消费者消费的offset位置信息  ./kafka-consumer-groups.sh --bootstrap-server c7node1:9092,c7node2:9092,c7node3:9092 --describe --group MyGroupId  #重置消费者组的消费offset信息 ，--reset-offsets –all-topics 所有offset。--to-earliest 最小位置。  # --execute 执行  ./kafka-consumer-groups.sh --bootstrap-server c7node1:9092,c7node2:9092,c7node3:9092 --group MyGroupId --reset-offsets --all-topics --to-earliest --execute |

1. SparkStreaming+Kafka
2. receiver模式

* receiver模式原理图



* receiver模式理解：

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。

* 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。

如何解决这个问题？

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。

* receiver模式代码（见代码）
* 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。

1. Driect模式

* Direct模式理解

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。

* 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。

* Direct模式代码（见代码）

1. 相关配置

预写日志:

|  |
| --- |
| spark.streaming.receiver.writeAheadLog.enable 默认false没有开启 |

blockInterval:

|  |
| --- |
| spark.streaming.blockInterval 默认200ms |

反压机制:

|  |
| --- |
| spark.streaming.backpressure.enabled 默认false |

接收数据速率:

|  |
| --- |
| spark.streaming.receiver.maxRate 默认没有设置 |

1. SparkStreaming2.3+kafka 改变
2. 丢弃了SparkStreaming+kafka 的receiver模式。
3. 采用了新的消费者api实现，类似于1.6中SparkStreaming 读取 kafka Direct模式。并行度一样。
4. 因为采用了新的消费者api实现，所有相对于1.6的Direct模式【simple api实现】 ，api使用上有很大差别。未来这种api有可能继续变化
5. kafka中有两个参数：

heartbeat.interval.ms：这个值代表 kafka集群与消费者之间的心跳间隔时间，kafka 集群确保消费者保持连接的心跳通信时间间隔。这个时间默认是3s.这个值必须设置的比session.timeout.ms appropriately 小，一般设置不大于 session.timeout.ms appropriately 的1/3。

session.timeout.ms appropriately：这个值代表消费者与kafka之间的session 会话超时时间，如果在这个时间内，kafka 没有接收到消费者的心跳【heartbeat.interval.ms 控制】，那么kafka将移除当前的消费者。这个时间默认是10s。这个时间是位于 group.min.session.timeout.ms【6s】 和 group.max.session.timeout.ms【300s】之间的一个参数,如果SparkSteaming 批次间隔时间大于5分钟，也就是大于300s,那么就要相应的调大group.max.session.timeout.ms 这个值。

1. 大多数情况下，SparkStreaming读取数据使用 LocationStrategies.PreferConsistent 这种策略，这种策略会将分区均匀的分布在集群的Executor之间。

如果Executor在kafka 集群中的某些节点上，可以使用 LocationStrategies.PreferBrokers 这种策略，那么当前这个Executor 中的数据会来自当前broker节点。

如果节点之间的分区有明显的分布不均，可以使用 LocationStrategies.PreferFixed 这种策略,可以通过一个map 指定将topic分区分布在哪些节点中。

1. 新的消费者api 可以将kafka 中的消息预读取到缓存区中，默认大小为64k。默认缓存区在 Executor 中，加快处理数据速度。可以通过参数 spark.streaming.kafka.consumer.cache.maxCapacity 来增大，也可以通过spark.streaming.kafka.consumer.cache.enabled 设置成false 关闭缓存机制。
2. 关于消费者offset

1).如果设置了checkpoint ,那么offset 将会存储在checkpoint中。这种有缺点: 第一，当从checkpoint中恢复数据时，有可能造成重复的消费，需要我们写代码来保证数据的输出幂等。第二，当代码逻辑改变时，无法从checkpoint中来恢复offset.

2).依靠kafka 来存储消费者offset,kafka 中有一个特殊的topic 来存储消费者offset。新的消费者api中，会定期自动提交offset。这种情况有可能也不是我们想要的，因为有可能消费者自动提交了offset,但是后期SparkStreaming 没有将接收来的数据及时处理保存。这里也就是为什么会在配置中将enable.auto.commit 设置成false的原因。这种消费模式也称最多消费一次，默认sparkStreaming 拉取到数据之后就可以更新offset,无论是否消费成功。自动提交offset的频率由参数auto.commit.interval.ms 决定，默认5s。如果我们能保证完全处理完业务之后，可以后期异步的手动提交消费者offset.

3).自己存储offset,这样在处理逻辑时，保证数据处理的事务，如果处理数据失败，就不保存offset，处理数据成功则保存offset.这样可以做到精准的处理一次处理数据。